SOFTWARE AND CRITICAL TECHNOLOGY PROTECTION AGAINST SIDE-CHANNEL ANALYSIS THROUGH DYNAMIC HARDWARE OBFUSCATION

THESIS

John Bochert, 1st Lt, USAF

AFIT/GCE/ENG/11-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT/GCE/ENG/11-01

SOFTWARE AND CRITICAL TECHNOLOGY PROTECTION AGAINST SIDE-CHANNEL ANALYSIS THROUGH DYNAMIC HARDWARE OBFUSCATION

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

John Bochert, BSEE
1st Lt, USAF

March 2011

AFIT/GCE/ENG/11-01

SOFTWARE AND CRITICAL TECHNOLOGY PROTECTION AGAINST SIDE-CHANNEL ANALYSIS THROUGH DYNAMIC HARDWARE OBFUSCATION

John Bochert, BSEE
1st Lt, USAF

Approved:

Dr. Yong Kim, PhD (Chairman)	14 MAR 2011 Date
Maj. Todd R. Andel, PhD (Member)	14 MAR 11 Date
Dr. Michael Grimaila, PhD, CISM, CISSP (Member)	10 MAR 2011 Date
Lt. Col. Jeffrey W. Humphries, PhD (Member)	10 Mar 2011 Date

AFIT/GCE/ENG/11-01

Abstract

Side Channel Analysis (SCA) is a method by which an adversary can gather information about cryptographic keys by examining the operations on a microprocessor measuring electromagnetic (EM) emissions or any environment changes the processor creates during the execution of a cryptographic algorithm.

The purpose of this thesis is to devise methods to increase the difficulty of conducting SCA successfully. This thesis makes use of the open-source soft-core Java Optimized Processor (JOP) on a Xilinx Virtex 5 ML506 evaluation board which is used for evaluating the effectiveness of SCA countermeasures in attacks against cryptographic algorithms. Analysis of SCA in attacks against RSA cryptography implemented on the JOP revealed that secret key information was leaked through the ALU and multiplier. In examining the processor in attacks against AES cryptography, the JOP was found to leak secret key information through memory accesses. To increase security this thesis proposes the use of several "double hardware" sets that are dynamically used in such a way to obfuscate the EM emissions to increase security against SCA attacks. In the case of RSA where the ALU and multiplier leaks SCA information, the proposed methods removed a correlation of the JOP emissions by using two separate adders and two separate multipliers. In each case, the adders and multipliers that were alternated were chosen due to their specific difference in power signature used while computing the sum or product, thus effectively obfuscating the bit-group exponentiation functions in Java. In the case of memory correlation and attacks against AES, the proposed methods normalized the power usage of stored values, so that all values being stored look the same. This method increased security by requiring up to 87% more data to successfully attack AES.

Acknowledgements

First and foremost, I owe my biggest debt of gratitude to God, for without His constant provision, I would be lost. I would also like to thank Dr. Kim, my advisor, who was always available for discussions of problems and solutions. He was able and willing to impart both knowledge and wisdom to help me accomplish my goals. I would also like to thank the members of my thesis committee (Dr. Grimaila, Lt Col Humphries, and Major Andel) who helped me see problems from different angles in order to solve them different ways. I would also like to thank all of the members of the VLSI group (Cpt Trejo, Cpt Getz, Lt Falkinburg and Lt Stanton) who were there to help me when they could, and there to laugh with (at) me when I made silly mistakes. Lastly, I would like to thank my family. Although far away, they were there for me when I needed someone to talk to.

John Bochert

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Abbreviations

Abbreviation		Page
SCA	Side Channel Analysis	1
ATMs	Automated Teller Machines	1
SSL	Secure Socket Layer	1
EM	Electromagnetic	3
API	Application Programming Interface	4
RSA	Rivest, Shamir, and Adleman	5
AES	Advanced Encryption Standard	5
DAILMOM	Dual Asynchronous In-Line Multiplexed Out/inputs Multiplier	5
RAM	Random Access Memory	6
EMSEC	Electromagnetic Security	7
CMOS	Complementary Metal-Oxide Semiconductor	7
HW	Hamming Weight	8
HD	Hamming Distance	8
SPA	Simple Power Analysis	8
DPA	Differential Power Analysis	8
SODPA	Second Order Differential Power Analysis	8
CPA	Correlation Power Analysis	9
RPI	Random Process Interrupts	12
RSL	Random Switching Logic	14
DVFS	Dynamic Voltage and Frequency Switching	14
MDPL	Masked Dual-Rail Pre-charge Logic	14

Abbreviation		Page
DRP	Dual-Rail Precharge	14
DDL	Dynamic and Differential Logic	15
SOP	Sum Of Products	18
JVM	Java Virtual Machine	18
ALU	Arithmetic Logic Unit	19
CRT	Chinese Remainder Theorem	22
JOP	Java Optimized Processor	26
GNU	Gnu's Not Unix	26
FPGA	Field Programmable Gate Array	26
ASIC	Application-Specific Integrated Circuit	26
CISC	Complex Instruction Set Computer	26
RISC	Reduced Instruction Set Computer	26
VHDL	Very high speed integrated circuit Hardware Description Language	27
JTAG	Joint Test Action Group	27
JDK	Java Development Kit	27
RCA	Ripple Carry Adder	29
CLA	Carry Look Ahead	29
SCIO	SimpCon Input/Output	34
SUT	System Under Test	41
CUT	Components Under Test	41
CUDA	Compute Unified Device Architecture	51
DES	Data Encryption Standard	85

SOFTWARE AND CRITICAL TECHNOLOGY PROTECTION AGAINST SIDE-CHANNEL ANALYSIS THROUGH DYNAMIC HARDWARE OBFUSCATION

I. Introduction

The introduction chapter begins with a brief introduction into the history of Side Channel Analysis (SCA) attacks to set the stage for the motivation of this thesis. After providing a brief history, this chapter outlines four types of attacks on microprocessors running cryptographic code, and explain which type of attack SCA falls under. At this point, the introduction chapter briefly explains what Side Channel Analysis is and provides a background for how it applies to microcontrollers. At the end of the chapter is the research problem statement, goals and objectives, proposed methods, and an outline of what the next four chapters will contain.

1.1 History

Security and cryptography in electronics have played an integral part in society for several decades. Starting in the 1970s with securing military communication channels and in the civilian sector with Automated Teller Machines (ATMs), the need for security has been on the rise since first being sold on the open market. Secure crypto-processors (microprocessors that process cryptographic algorithms) will often be the backbone of security networks like in the case of ATMs. Crypto-processors have also found more widespread use in protecting Secure Socket Layer (SSL) keys and defending proprietary software and algorithms from theft, both from the outside world and from employees. Cryptography has become essential to our society. One can find crypto-processors in smart cards, decryption keys for pay TV, lottery ticket vending machines, and mobile-phone systems. For all these crypto-processors, companies deploy various means of tamper resistance to prevent system tampering, reproduction, disabling, and reverse-engineering [4].

In the 1970s when crypto-processors were first being used, there was only a need for simple security measures and tamper-resistant hardware. IBM developed the 3614 ATM system for authenticating customers at ATMs with a bank-given PIN. Although it did not employ significant anti-tamper protocols, it contained various algorithms to prevent whole-sale discovery of PINs and other simple methods to prevent employee tampering [5]. In time, cryptographic systems have become more complicated, and their security has greatly increased. The IBM 3848 was one of the first crypto-processors to incorporate an active anti-tamper mechanism. It was enclosed in a steal case that would zero out the memory when it was opened. Its successor, the IBM 4758 (see Figures 1 and 2) was a cryptographic engine surrounded by a multilayer tamper-sensing mesh. The mesh was constantly monitored and would zero out all key material if tampering was sensed [10].

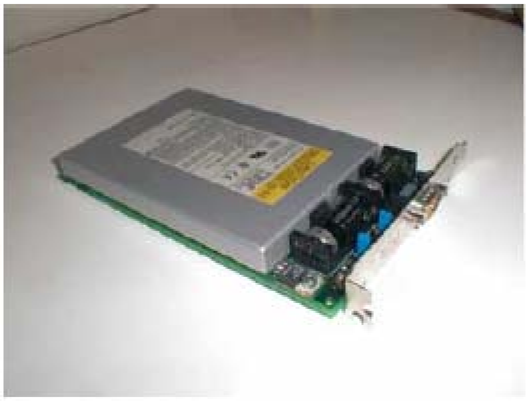


Figure 1. IBM 4758-001.

1.2 Motivation

1.2.1 Attack Types.

There are four different classes of attack by which an adversary can attack a crypto-processor: Local Noninvasive Attacks, Semi-Invasive Attacks, Invasive Attacks, and Remote Attacks [4]. Local Noninvasive Attacks involve gaining information about the device through close observation of the device in operation, watching both the Electromagnetic (EM) emissions and various environmental information around the device. Local Noninvasive Attacks was chosen to be the focus of this research because of it's importance because it allows any attacker to circumvent cryptographic algorithms just by having access to the device, and the owner of the crypto system may never even know about the attack. Additionally, although not covered by this thesis, are Semi-Invasive Attacks, Invasive Attacks, and Remote Attacks. Semi-Invasive Attacks do not require damaging of the passivation layer, and instead use lasers to ionize transistors and change its state. This method is difficult to use to extract information due to the variability inherent in trying to ionize specific transistors. Invasive Attacks involve actual damage to the device and connection or monitoring of the device interior. Although this is a useful, because it destroys the device there are large risks associated with this method. Remote Attacks, as in the case of an

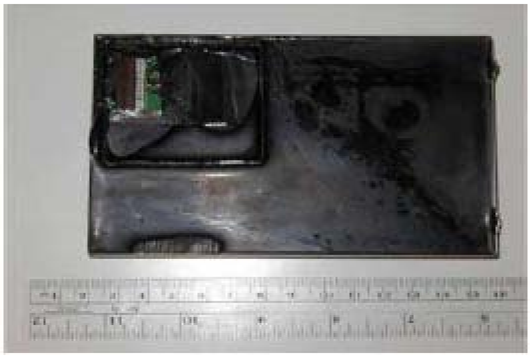


Figure 2. An IBM 4758-001 part potted in urethane, showing membrane and interior.

Application Programming Interface (API) like SSL, are done remotely and interface with a device in normal operation. These methods are useful, but they deal solely with programming vulnerabilities and not hardware vulnerabilities [4].

SCA attacks, also known as Local Non-Invasive attacks, are the method by which an adversary can cleverly deduce information about a cryptographic system by watching the interaction of a digital circuit with its surrounding environment. The research is focusing on the attacks because this method allows an adversary to circumvent a secure cryptoalgorithm just by having physical access to the device without the device owner ever knowing their crypto system was compromised. The three most important types of SCA are timing, power-analysis, and EM attacks. In all types, the basic idea is to determine a cryptographic device's secret key by measuring its execution time, power consumption, or electromagnetic field [30].

1.3 Problem Statement

This research addresses the problem of SCA for the cryptographic algorithms, where a cryptography key can be made known to an adversary through close monitoring of the device in operation. Furthermore, this research aims to reduce the susceptibility of cryptography systems to SCA attacks.

1.4 Goals and Objectives

The goal of this research is to propose new methods to protect cryptographic information and critical technology by making dynamic changes to the underlying architecture of a microprocessor. The objectives of this research are to implement SCA vulnerable algorithms, compromise them, and then show an increase in the security of the circuit after incorporation of the proposed secure hardware.

1.5 Proposed Protection Methods

The proposed methods of this research are derived from the hypothesis that by adding multiple hardware units that are functionally the same but structurally different, the EM emissions off the microprocessor can be varied increasing the difficulty to conduct SCA successfully. There are two proposed methods to protect the cryptographic algorithm Rivest, Shamir, and Adleman (RSA) and two proposed methods to protect the cryptographic algorithm Advanced Encryption Standard (AES). To protect RSA, the two proposed methods are a Double Adder and a Dual Asynchronous In-Line Multiplexed Out/inputs Multiplier (DAILMOM). These two proposed countermeasures obfuscate RSA where the algorithm is most vulnerable to SCA attacks, and are used to override the current addition and multiplication functions on the microprocessor. The Double Adder increases the security by alternating the power signature of the add operation each time an addition is called. This is done by using a Carry Look Ahead adder whose power signature to add numbers is front-loaded, and the Ripple Carry Adder whose power signature is evenly distributed during addition. Alternating which adder is used allows for one add to have a front-loaded or an even distribution during addition, and then the next addition operation, regardless of the values being added, will have the other power signature. The DAILMOM uses two multipliers and it's obfuscation is due to the result of the second multiplier being fed arbitrary data that will shape the power signature a different way each time. This overlay of noise whose signature can be manipulated allows for two subsequent multiply operations to have the same data but totally different power signatures. To protect AES, the two proposed methods are a Double Stack and a Double RAM. These two proposed countermeasures aim to protect AES where it is most vulnerable to SCA attacks by changing the way the microprocessor stores values on the stack during the execution stage and how values are saved to the Random Access Memory

(RAM) during the writeback of variables. This increase in security is due to when a variable is being saved, it is split into two variables, where each resulting variable being saved always has the same number of logic '1's and '0's. This power normalization allows for the data to always have the same usage, decreasing the power changes in the circuit that SCA looks for.

1.6 Research Methodology

Testing of the proposed methods will be done by implementing both RSA and AES on a microprocessor being implemented on an FPGA. First, the two cryptographic systems will be evaluated for their initial security against SCA. After a baseline security is established, the microprocessor security will be reevaluated with the proposed methods and be recompared to the original microprocessor.

1.7 Organization

Chapter 2 provides background information on current research with SCA and protection against SCA. Chapter 3 details the approach for implementation and evaluation of the stated objectives. Chapter 4 details the results of the experiments and any adjustments required to facilitate the thesis goals. Chapter 5 evaluates the work undertaken and reviews the major contributions of this thesis, which includes future directions for extending this research.

II. Related Work

Chapter 2 begins by explaining Differential and Correlation Power Analysis. Chapter 2 then explains the current research in preventing SCA attacks separated into algorithmic countermeasures and circuit level countermeasures. After this, the thesis examines the cryptography algorithms Rivest, Shamir, and Adleman (RSA) and Advanced Encryption Standard (AES) and then goes through the JOP in detail.

2.1 Differential and Correlation Power Analysis

Beginning in June of 1998 with Kocher's initial publication of the susceptibility of cryptosystems to side channel analysis [13] and his follow up article [14], much research has been done to both protect and exploit circuits. Although SCA attacks have become more sophisticated in the last decade, the basic idea of SCA is not new. For example, military circles were aware of the need for EM security since the discovery of crosstalk between telegraph lines of the British army expedition to the Nile and Suakin in 1884-1885. Even in World War I, telephones on the front lines could be eavesdropped by the opposing side at distances of hundreds of yards. Nowadays, "Tempest" military standards help to govern electronic emissions for military devices and for Electromagnetic Security (EMSEC) [1].

2.1.1 Background.

The basic premise of SCA attacks stem from the reality that the switching activity of Complementary Metal-Oxide Semiconductor (CMOS) circuits leak information. When a CMOS circuit charges to a '1' or discharges to a '0', a change in the electric potential creates a change in the electric field which is measurable outside the chip. Generally the quantization of the energy for a given value is derived from either the

Hamming Weight (HW) or the Hamming Distance (HD). In the case of the HW, the value of a given data is the summation of the bits that are in a ‘non zero’ state. That is to say that the HW of 0x50 (0b01010000) is two, and the HW of 0x03 (0b00000011) is also two while the HW of 0xFF (0b11111111) is eight. The HD however is a measure of the change of a value, measuring the number of bits that change from the previous state to the current state. For example the hamming distance between 0x50 and 0x03 is 4, while the hamming distance between 0x50 and 0xFF is 6. The Hamming Weight can be thought of as the Hamming Distance between the zero value 0x00 and any other value. Commonly, the model used to describe the information leakage off a chip is $C(t)^{(a,b)} = \lambda HW(a \otimes b) + \beta_t$, where $(a \otimes b)$ is the XOR of a and b, HW is the Hamming Weight function, λ is the power consumption used by the circuit when inverting the bit, and β_t is noise [6].

2.1.2 SCA Attack types.

From monitoring the execution time, power consumption and the electric field from a microprocessor, the three main types of SCA attacks used to find secret key information are: Simple Power Analysis (SPA), Differential Power Analysis (DPA), and Second Order Differential Power Analysis (SODPA) [28]. A SPA attack involves directly observing a system’s power consumption. This attack is particularly useful when an adversary only has the ability to capture only one encryption because SPA can be done on a single trace. If the attacker knows generally what the processor is doing, an attacker can visually analyze the trace and extract information. This attack is also particularly useful in RSA, where binary exponentiation exacerbates visual timing differences that depend on the key in the power trace.

DPA is significantly more powerful than SPA, but is more complicated and requires many more traces. For DPA to work, one needs to know either the plaintext or ciphertext of each trace. The known plain/ciphertext of each trace is then used with hypothesis testing for the first bit of the key. From this hypothesis and using the knowledge of the plain text for each trace, all of the traces can be separated into two pools: if the resulting bit of the plain text after encryption is a '1' or a '0' given the key hypothesis [20]. With these two pools, statistical analysis can be conducted to search for correlating power biases in the comparisons that will arise for correct key guesses. From these power biases, the correct key can be derived.

Lastly, SODPA is the most powerful SCA tool, but also the most complicated. SODPA is a method employed to overcome the masking countermeasure employed by some SCA defenses. In masking, a defender splits a value Z into d shares $M_1 \diamond ... \diamond M_d$ such that $M_1 \diamond ... \diamond M_d$=Z and where $\diamond$ is a function like the XOR or modular addition [23]. A masking operation is said to be (d-1)th-order depending on the number of shares d. When a (d-1)th order masking is used, a dth-order DPA can be performed by combining the leakage signals at time intervals L(t_1), ... , L(t_d) resulting from the manipulation of the d shares that make up the value Z. The downside to this attack is that noise effects in the leakage current readings increases exponentially with the number of d shares. Currently there are no designs of higher order masking that are efficient and secure beyond the second order, as such only second order masking is generally used in the protection of block ciphers.

2.1.3 Differential Power Analysis vs Correlation Power Analysis.

There are namely two methods referred to as Differential Power Analysis, the original DPA (also known as the Difference in Means Method) and Correlation Power Analysis (CPA) [11]. The original DPA was the first method used by Kocher in his attack of the cryptographic system. In this attack, one feeds a known plain text to the cryptographic system (or keeps track of the cipher text and works backward from the traces instead) and records the power trace that corresponds to that data. In order for this attack to work, one needs many traces with many different known plain texts. Then the attacker works down the bits of the key, one by one, making a hypothesis guess as to if that bit is set or not individually. Having done this, the attacker separates all of the traces into two groups, those who the key guess would result in them having '0' for the given value after encryption, and those that the key guess would result in them having a '1' after encryption. Once the traces have been separated, the attacker averages all the traces for each group, and then find the difference of the means. If your key guess was correct, then the difference of means trace results in a power spike. If the key guess was incorrect, then the resulting difference trace would not contain any spikes. This method of separating traces into bit groups, finding the average trace, and then finding the difference in means can be seen in Figure 3 [25].

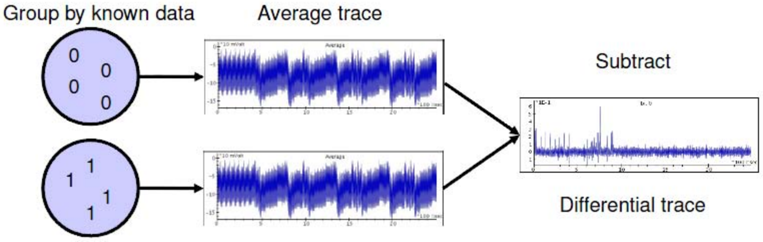


Figure 3. Differential Power Analysis

Though often bundled up with traditional Differential Power Analysis under the name of DPA, Correlation Power Analysis is a newer method for power analysis and computes keys in a different fashion. In the case of CPA, linear algebraic methods are used to find keys in the traces. Given many traces with many plain texts, the correlation coefficient is a measure of the linear relationship of the plain text with the trace value at a given instant in time. The correlation power trace is derived by using Pearson's coefficient of correlation on the 'data' values and the 'trace' values for each time instant, and then stepping in time down the trace and finding the resulting coefficient of correlation for each time instant. Figure 4 shows an example of CPA being used on four traces where the data (FF, DD, 09, and 8B) is being correlated with the values in the first red box to create the first value of the 8-bit correlation. The subsequent red boxes attempt to show the correlation being done at every time instant to create the full 8-bit correlation that is examined for spikes [11] [25].

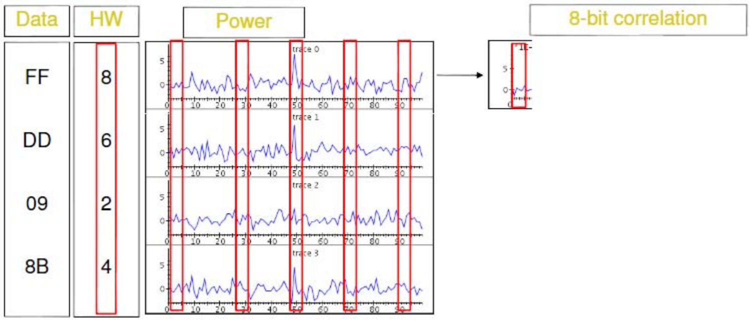


Figure 4. Correlation Power Analysis

In practice, Correlation Power Analysis' use of linear algebra actually has a significant reduction in the effect of random noise in power traces. So using CPA instead of traditional DPA results in the correct key with fewer traces. For the remainder of this document, when DPA is used on a given sample set, CPA is the method of DPA employed unless otherwise stated. As explained in Chapter 3, these methods of power analysis attacks are accomplished on the data using the third party software, Riscure Inspector.

2.2 Algorithmic Countermeasures

As mentioned earlier, SCA attacks fit into three categories - Simple Power Analysis, Differential Power Analysis, and Second Order Differential Power Analysis. Defenses against these attacks fit into two high-level categories: algorithmic countermeasures and circuit-level countermeasures. Countermeasures can be further classified based on the method by which they try to decouple the power consumption with the data being processed, these are: masking countermeasures and elimination (hiding) countermeasures [28].

2.2.1 Masking Techniques.

Algorithmic countermeasures try to uncorrelated the output of algorithms with the secret key. Masking at the algorithmic level has the key notion of minimizing the correlation between intermediate values and the secret key [7]. One simple method to accomplish this is to introduce noise into the power consumption measurements. This method is overcome by the introduction of more samples. In theory if the variance of the noise is great, then the necessary sample size might be infeasible, but this method is still surmountable by increasing the number of samples [15].

Another option to the end of masking power traces at the algorithmic level is the introduction of Random Process Interrupts (RPI) during the cryptographic algorithm. This approach can be done by interleaving random dummy commands or "No Operation' (NoOp commands)' randomly throughout the code thus masking the actual execution sequence. Thus RPIs cause time shifts in the corresponding cryptographic operations and thus mismatch amongst them. This mismatch reduces the size of the correct differential spikes by spreading them over several clock cycles in different traces. It also causes an increase in the number of clock cycles needed for analysis proportional to the squared number of the clock cycle variation that occurs. That is to say, if k is the number of clock cycles that the event varies, then the number of necessary traces is proportional to k^2. Although this increase in the required clock cycles can make a SCA attack infeasible, it does not prevent an attack because the information is still being leaked. In the case of RPIs, the correlation spikes can be reconstructed though by integrating the signal over the number of consecutive clock cycles equal to the greatest number variance in the clock cycles [8]. This method to overcome RPIs is called the sliding window attack. For this attack, several traces are integrated together and then compared against other integrated traces for the power spikes [17].

Another method for algorithmic masking is to split the key into two halves (duplication method) [9] or more generally, by splitting the key into k shares [7]. However, this method of masking can be circumvented through the use of higher-order differential power analysis. By combing the leakage signals at time intervals L(t_1), ... , L(t_d) that are the resulting leakages from the manipulation of the d shares that make up the value Z, the differential power spike for correct key guesses can be reproduced [23][19][21].

2.2.2 Elimination Techniques.

Elimination is the other method that can be used to confound power variation. The key notion of elimination (hiding) is to remove power variation information from the attacker. Where masking seeks to decouple the power variation from the data being processed, elimination seeks to eliminate it. Four ways that people employ to use elimination are: [15]

1 - Using constant execution path code

2 - Choosing operations that leak less information in their power consumption

3 - Balancing hamming weights and state transitions

4 - By physically shielding the device

2.3 Circuit Level Countermeasures

2.3.1 Masking Techniques.

Circuit level countermeasures to remove the correlation between a circuit's power characteristics and the data being processed. In CMOS circuitry, power is consumed by the system when an output transitions from a logical 0 to a 1. These power spikes are what an adversary uses in power traces to gleam information. Many masking techniques at the circuit level introduce random power consumptions which are akin to noise. Examples include Random Switching Logic (RSL) [29], masking-AND [34], and Dynamic Voltage and Frequency Switching (DVFS) [35]. The RSL countermeasure adds in random logic paths, masking-AND masks every output with random inputs, and DVFS randomly modulates voltage and switching frequency to introduce randomness into power traces. All of these circuit level masking techniques, however, are still susceptible to glitches. Glitches are the transitions at the output of a gate that occur before the gate switches to the correct output. Because glitches add to the power signature, they are susceptible to leak information, especially when they leak key information before the correct mask is applied [18][3]. RSL uses random input and enable control signals to randomize the power signature and is thus able to avoid the information leakage posed by glitching, but the enable signals need to be carefully timed [3].

A more advanced method for circuit level masking however is Masked Dual-Rail Pre-charge Logic (MDPL) [22]. MDPL uses both masking and dual rail precharged logic to obfuscate the power signature. The masked values are represented as $bm = b + m$, where b is the actual value and m is the applied mask. Furthermore, the mask, m, is a bit generated randomly and updated every clock cycle. The Dual-Rail Precharge (DRP) logic addresses the glitches that occur in CMOS circuitry by precharging all signals to 0 before every evaluation phase and thereby alternating between the precharge and evaluation states. MDPL also makes use of majority function gates which only change the output once during the precharge phase and once during the evaluation phase. An example of the MDPL AND gate is shown in Figure 5.

The majority function gate goes high when most of the inputs are high, and goes low when most of the inputs are low. The six dual-rail inputs can be seen in the picture as well: a_m, b_m, c_m, a'_m, b'_m, and c'_m. The dual rail logic claims to have equal capacitances and thus prevents glitching by evaluating both the logic value and its inverse simultaneously. Although the precharge logic aims to prevent glitching, it also has the added benefit of helping to smooth out power use [22]. However, with all these advances, [16] shows that MDPL is susceptible to leakage due to early propagation, which is when a gate calculates the output before all the inputs arrive. Further, [26] shows loading balances that occur due to a lack of routing constraints which can be used to expose a DES key. The mask bit is also fairly easy to remove by examining the trace and seeing if the power trace is above or below the average power consumption. Furthermore as an efficiency note, the necessary power for MDPL increases by more than double due to the dual rail logic and the added mask circuitry and precharging.

2.3.2 Elimination Techniques.

Elimination, as mentioned earlier, attempts to eliminate power variation by making the power consumption of the device constant for any input data. Without any variation in power due to the key, no information is leaked through side channels. Constant power can be achieved by ensuring that a constant power is consumed every clock cycle. This is done in the Dynamic and Differential Logic (DDL) style by ensuring that one of the outputs is charged for any input, be it the output or the complimented output, and it ensures that one output transition occurs in every clock cycle. More specifically, DDL logic is split into a precharge state, where all outputs are at zero, and then an evaluation phase, where at least one output or its compliment goes high [31]. Sense Amplifier Based Logic is an implementation of DDL that uses a dynamic-CMOS logic implementation. Using this implementation of DDL, however, causes Amplifier Based Logic to have to deal with the effects of cascading circuits and signal integrity issues that derogate the signal and make it inefficient [31].

Wave Dynamic and Differential Logic (WDDL) is another implementation of DDL. WDDL uses a static CMOS implementation of AND and OR gates. Each gate in the WDDL has both the gate with the inputs and a complimentary gate with the inverse of the inputs. For example, Figure 6 shows a WDDL and gate with its logic table where "Prch" is the precharge state that the gate goes into between evaluations, and the "Eval" state is the evaluation of the gate given the inputs shown in the figure.

With the shrinking capacitor size, however, it can be seen that the signal propagation capacitance is actually greater than the transistor capacitance so to be truly effective at stopping SCA the output lines need to be placed on the board side by side with the same length of wire. Also with this logic, early propagation can be a problem which needs to be addressed. The WDDL also uses only positive gates, so that the precharge phase can propagate through all gates, which makes logic optimizing difficult. Lastly, to contain all the logic and its compliment would require a cryptosystem implementing WDDL to have at least twice the area and require twice the power as a cryptosystem that does not implement WDDL [32]

Reduced Complementary Dynamic Differential Logic (RCDDL) is another implementation of DDL. Like WDDL, RCDDL has both the logic and its complimentary logic, but RCDDL seeks to optimize its logic while keeping its security features. To optimize, RCDDL reuses logic gates in the uncomplimented logic while ensuring that a constant capacitance is charged. Unlike WDDL, RCDDL allows the both positive logic (AND/OR) and negative logic (NAND/NOR). The RCDDL gate structure can be seen in Figure 7.

As can be seen in Figure 7, the uncomplimented output, Y, is calculated through Segments I and II. Segments I, III, and IV collectively form the complimented out-

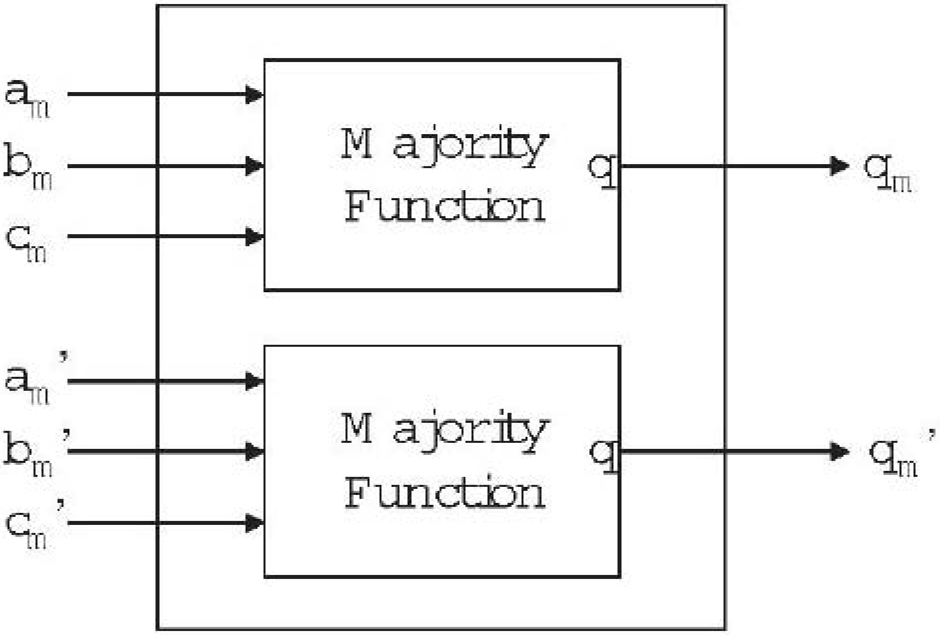
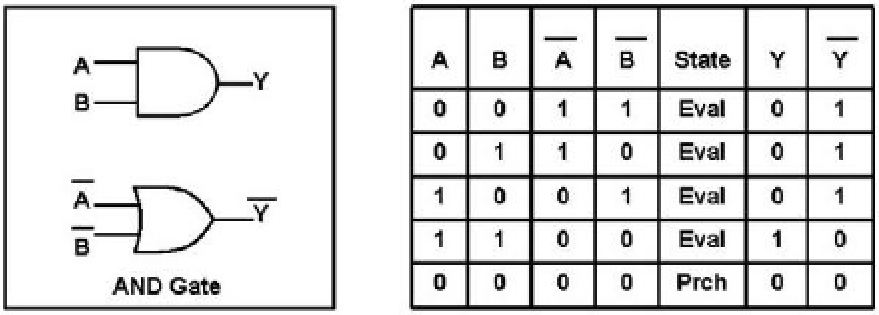


Figure 5. MDPL AND Gate [28]



A	B	$\overline{A}$	$\overline{B}$	State	Y	$\overline{Y}$
0	0	1	1	Eval	0	1
0	1	1	0	Eval	0	1
1	0	0	1	Eval	0	1
1	1	0	0	Eval	1	0
0	0	0	0	Prch	0	0

Figure 6. WDDL AND gate and truth table [28]

put, Y'. The segments are created through using the Sum Of Products (SOP) where Segment I is all but the last step, which is completed in Segment II. Segment III generates the precharge signal which sets IV to drive the complimented output to '0'. This is done locally in every gate, which does away with the need for a precharge phase thus allowing for RCDDL to use negative logic. Because of the reuse of gates with RCDDL, the size and power consumption of RCDDL are smaller than that of WDDL. Additionally, it has been shown that RCDDL has less power variation than WDDL thus making it more secure. Unfortunately, as compared to WDDL, due to the size of the individual RCDDL cells, there is a delay penalty. Early propagation effects also affect RCDDL so there is that added vulnerability. [24]

Although all DDL elimination strategies reduce the power consumption variation, in reality, all methods still have minor power variations. The goal of circuit-level elimination SCA countermeasures is to further minimize with the hope of eventually eliminating these variations [28].

Lastly, in his article [12], Slaman obfuscated EM emissions off the FPGA by changing Java bytecodes. This method had the added benefit of changing both the power and time correlation during code execution. The Java bytecode method of obfuscation however differs from this thesis in that it was all software implemented with no changes to the hardware of the JOP, and was thus written for the Java Virtual Machine (JVM) to protect java code running on the JOP. This thesis however proposes lower level changes by changing the actual hardware, allowing for a wider range of software security.

2.4 Cryptographic Algorithms

Two of the most common algorithms used in the military and in the civilian sector for encryption are the RSA algorithm developed in 1977 and the AES algorithm developed in 2001. In order to better protect the crypto-processor running RSA or AES against SCA attacks, this thesis employs the methods of both masking and elimination techniques to obfuscate the Arithmetic Logic Unit (ALU), the multiplier, and the memory to protect the circuit from EM and power SCA. Before explaining the proposed methods, it is important to first explain in greater detail both cryptography algorithms to better show why certain methods were chosen for obfuscation.

2.4.1 AES.

The AES algorithm is a symmetric key crypto-algorithm, using the same secret key to do both the encryption and the decryption of data. The AES flow can be seen in Figure 8.

As can be seen in Figure 8, there are three main segments of the AES algorithm, the initial round, the nine rounds in between, and then the final round. Each of these three segments are made up by the four functions: SubBytes, ShiftRows, MixColumns, and AddRoundKey. SubBytes uses a simple substitution algorithm and takes the current text hex values and substitutes the values with known quantities in a "substitution box" (also known as the Sbox). ShiftRows is a function where the data, oriented in a 4x4 matrix, has each of the four rows shifted where the shift amount is different for each row. MixColumns is the function where each column is treated as a four term polynomial and is multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ effectively resulting in the multiplication shown in Figure 9.

Finally, AddRoundKey is the function where the key, modified slightly for each round, is added to the current text [2].

Of those four functions, only AddRoundkey actually manipulates the data based on key, which makes AddRoundkey the important target for key extraction. Each

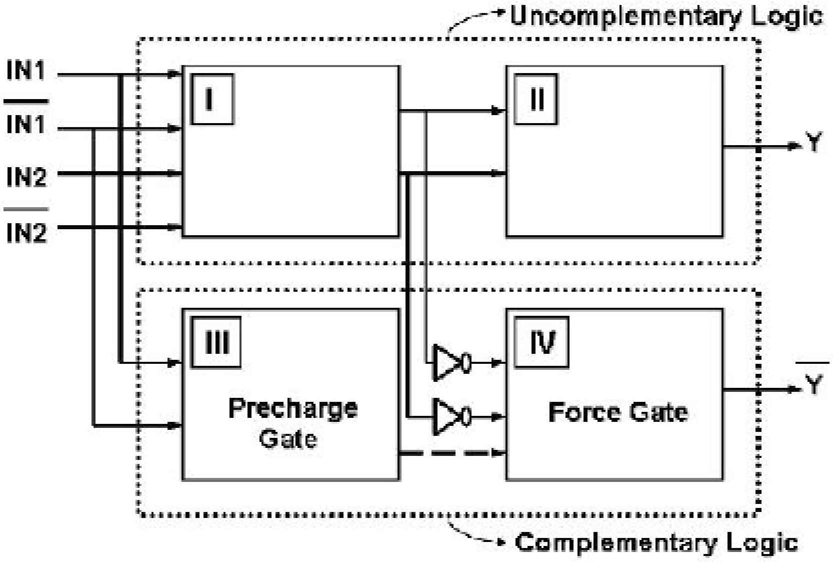


Figure 7. RCDDL Gate Structure [28]

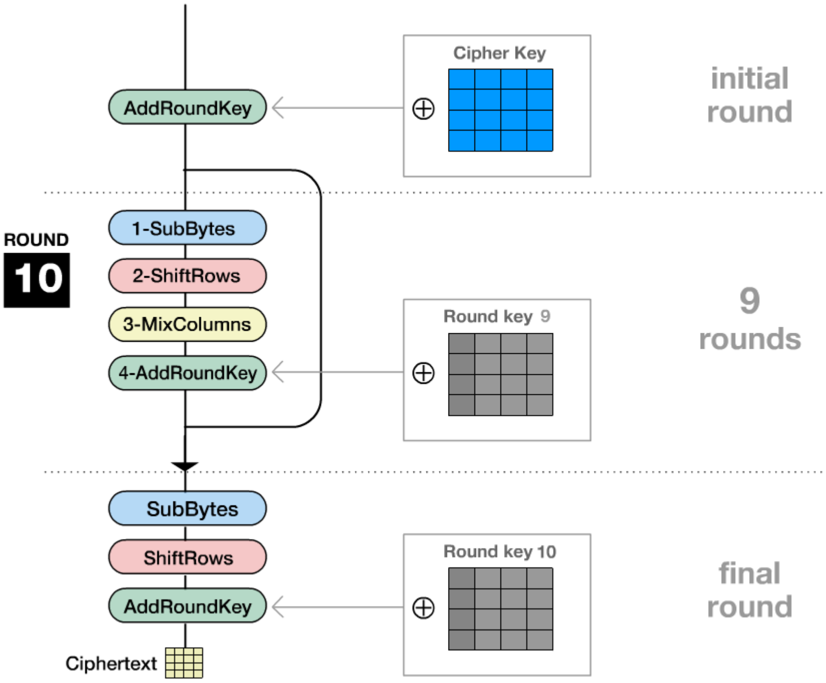


Figure 8. AES flow diagram[36]

time the function AddRoundKey is called, a different permutation of the key is used, with the first call to AddRoundKey using the original key, and each subsequent call to AddRoundKey using a different version of the key. The second function of importance is SubBytes. SubBytes is a simple substitution algorithm where by the current state of the plain text is used to find the corresponding substitution value in the Sbox. The Sbox is predefined and can be seen in Figure 10. For the SubBytes function, each byte in the current text is evaluated such that the first nibble of the byte corresponds to the row and the second nibble of the byte corresponds to the column. For example, if the byte being considered is $0x19$ then the value $0xd4$ will be substituted in for the value $0x19$. After these first two functions run is when the data is most susceptible to SCA attacks because the known plain text was just XORed by the key and the resulting state undergoes the Sbox substitution where many transistor transactions occur, thus leaking significant information of the current state of the plain text. Therefore, directly after the first SubBytes is the location of the SCA attacks on AES for this thesis.

2.4.2 RSA.

RSA is used in many areas, like in the area of e-commerce where RSA is one of the algorithms used to ensure safe transmissions of data over the internet. For example, a web site like Amazon is able to publish a public key for users all over the world to use but keeps a private key secret. In this manner users are able to encrypt sensitive

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 01 & 01 & 02 & 03 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}$$

Figure 9. MixColumns function

information like credit card numbers and pass the information over the internet to Amazon without fear that an eavesdropper would be able to decrypt the information.

RSA is known as an asymmetric key algorithm because there are two keys, one private and one public. This cryptography scheme with a public and a private key is made secure through difficulty of factoring numbers, but this leaves RSA only computationally secure but not provably secure. The computational security of RSA also means that the security of current RSA keys is based on the inherent difficulty in factoring a public key to get the private key. In theory, the modulus and the exponent (the public key, seen in Table 1 as e and n) has all the information needed to find the private exponent, but due to the difficulty in factoring a very large number, it is normally infeasible to find the private key using the public key. Keys for RSA are chosen based on them being infeasibly large to crack in a polynomial bounded computational time. As technology increases in complexity and power, likewise the key length of “secure” RSA implementations must also increase.

RSA is an asymmetric algorithm and is very different to attack than AES. The algorithm used by RSA can be seen in Table 1. As previously mentioned, the key to the secure mathematics behind RSA is based in prime number theory, enabled by the two primes p and q as seen in Table 1. The important step for consideration though for gaining the key in a SCA attack is the fifth step, where $c \equiv m^e (mod\ n)$. Unlike AES, RSA is completed with one basic function, a power mod function, where a large number can efficiently be raised to a large power and mod by another large number. When implementing RSA, there are namely three methods by which this is done, binary exponentiation, the Chinese Remainder Theorem (CRT), and bit group exponentiation[25]. Binary exponentiation is the simplest method to efficiently raise a large number to another large number. As can be seen in the algorithm shown in Table 2, the algorithm goes through each bit of the exponent and squares the current

Table 1. RSA algorithm[33]

The RSA Algorithm

1. Bob chooses secret primes p and q and computes $n = pq$.
2. Bob chooses e with $gcd(e, (p-1)(q-1)) = 1$.
3. Bob computes d with $de \equiv 1(mod(p-1)(q-1))$.
4. Bob makes n and e public, and keeps p, q, d secret.
5. Alice encrypts m as $c \equiv m^e(mod\ n)$ and sends c to Bob.
6. Bob decrypts by computing $m \equiv c^d(mod\ n)$.

Table 2. Binary Exponentiation[25]

Binary Exponentiation Algorithm for $M = c^d$

```
(di is exponent bits $(0 \leq i \leq t)$)
1. $M = 1$

2. For i from t down to 0:
3. $M = M * M$
4. If $di = 1$, then $M := M * C$
```

number and if the key is a ‘1’, then it also multiplies it by the base number. Using the method outlined in Table 2 yields traces that easily show to the naked eye (Simple Power Analysis) what the value is of the exponent. Although in cryptographic circuits, this algorithm is easily adapted to obfuscate the key value by always doing a square or a multiply. For example, squaring the value M and storing the result as variable X, and then multiply X by the base value C and store that in variable Y. Then if the bit is a one, use Y in the next stage, if it is a zero, then use X. This causes the attacker to need to use more advanced methods to discover the key if binary exponentiation is use.

The second method by which the RSA algorithm can be efficiently implemented is using the CRT, as can be seen in Table 3. In the case of using the CRT for the exponentiation algorithm, simple power analysis is not an option, although this method is still vulnerable to SCA attacks. Because CRT is highly efficient and quick,

Table 3. Chinese Remainder Theorem[25]

```
Chinese Remainder Theorem Algorithm for $M = c^d$
1. Precompute
dp = d mod (p-1)
dq = d mod (q-1)
K = p-1 mod q
2. Reduction
Cp = C mod p
Cq = C mod q
3. Exponentiation
Mp = Cp
dp mod p
Mq = Cq
dq mod q
4. Recombination
M = ( ( (Mq - Mp)*K ) mod q ) * p + Mp
```

it is highly appealing to people implementing RSA. In terms of attacking CRT, the attacker is actually trying to find the prime number that created the public key in the first place. To implement CRT, one uses the original prime number p and q in its evaluation, so the attack of the CRT is centered around finding the value of one of the primes. In the case of the algorithm below, in the final recombination step, the prime number p is isolated enough such that through the correlation of many traces, one can find p. Once the attacker has found p, he needs only to find q using the public modulus n and then he can use the public key to derive the private key mathematically.

The final method to implement RSA is bitgroup exponentiation. This case is similar to binary exponentiation, except that the exponent is broken into k-sized groups of bits and evaluated k bits at a time. For example, in a case of bitgroup exponentiation with a bitgroup of three, the resulting operations can be seen in Table 4. Prior to evaluation of the exponents, the values of c^1, c^3, c^5 and c^7 are evaluated

Table 4. Bit Group Multipliers for a bit grouping of 3, $M = c^d$

Exponent Bit Group	Resulting Operation
0	square
1	square, multiply by c^1
10	square, multiply by c^1, square
11	square, square, multiply by n^3
100	square, multiply by c^1, square, square
101	square, square, square, multiply by c^5
110	square, square, multiply by c^3, square
111	square, square, square, multiply by c^7

to save time and then used when needed. A three digit window then slides down the exponent and actions done to the value are dictated by the window.

Since RSA uses large numbers, the Java implementation of RSA utilizes a class called "BigInteger." In this class, the evaluation of $M = c^d modn$ is accomplished by calling the function "modpow(exp, mod)". This function then looks at the modulus and if it is odd, it calls the function "oddmodpow(exp,mod)" to evaluate $M = c^d mod\ n$. The function Oddmodpow impliments bit-group exponentiation with a big group size of three. In the case of RSA, because the value of the modulus is the product of two prime numbers, it will always be odd so when attacking the RSA implementation on the JOP, methods to crack the bit-group exponentiation are used to find the key. Interestingly however, if the modulus is even, Java splits the modulus into an "even" and an "odd" part using the CRT to break them apart, and then evaluates the "even" number with binary exponentiation, and evaluates the "odd" part with oddmodpow. Both "modpow" and "oddmodpow" are provided in Appendix A.

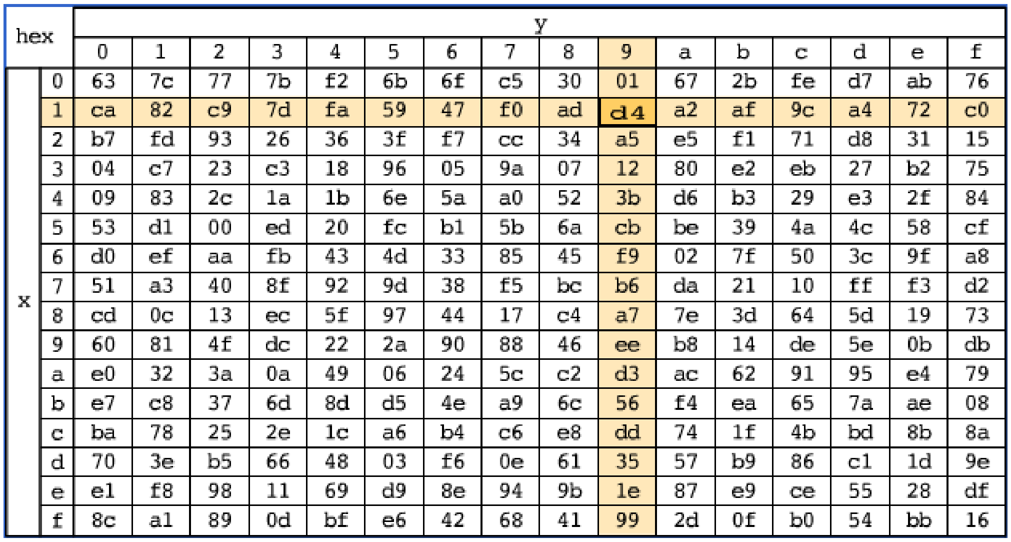
2.5 Java Optimized Processor

The main purpose of this thesis is to propose specific changes to the architecture of a microprocessor to make code running on the microprocessor more secure. Thus, cryptography code that was previously vulnerable to side channel analysis attacks can now be made more secure without changing the code but by using a modified version of the architecture. For the purposes of testing the security of the proposed thesis countermeasures, the Java Optimized Processor (JOP) was selected to implement the changes in the architecture and test the resulting security increases. The JOP is an open source soft-core microprocessor created by Martin Schoeberl and is published online at under the Gnu's Not Unix (GNU) General Public License, version 3. The JOP was chosen to be the vehicle for testing because it is a flexible, open source project. This allows easy changes to the architecture in the specification files, allowing these countermeasures to be implemented. The other reason the JOP was selected was that it is a soft-core processor that is able to be compiled and downloaded onto a Field Programmable Gate Array (FPGA). The use of an FPGA as opposed to an Application-Specific Integrated Circuit (ASIC) allows the flexibility needed to test implementations of these designs quickly and efficiently. The JOP main hardware open source core can be acquired online at [27].

The JOP is a 32-bit microcontroller that has been created by Martin Schoeberl as part of the PhD program at Vienna University of Technology, Austria. It was originally created to be an efficient method to implement Java on an FPGA, and has found a niche in many production level programs requiring the use of Java code outside of academia. The JOP is a stack based computer that has its own instruction set called "microcode." The JVM is a Complex Instruction Set Computer (CISC) like architecture which is turned into a Reduced Instruction Set Computer (RISC) like architecture on the JOP. The JOP block diagram can be seen in Figure 11. As can be seen from the JOP block diagram, the typical implementation of the JOP contains the JOP core, a memory interface and a number of I/O devices. The processor core contains the stages microcode fetch, decode, and execute with the additional bytecode fetch which is a translation stage. The extension module provides the link between the processor core, and the memory and I/O modules. It also controls data read and write. The memory interface provides a connection between the main memory and the processor core. It also contains the method cache. The control bit synchronizes the modules as the memory access is done concurrently with the JOP core instruction execution.

The first stage in the JOP core is the Bytecode Fetch stage. This stage is not actually part of the "JOP pipeline" and functions as the method where by Java bytecode (the JVM instructions) are translated into corresponding microcode addresses. After this translation stage, the bytecode equivalent instruction is then passed to the Fetch stage. This stage then acts like a regular Instruction Fetch stage in a pipelined microprocessor. Once the microcode has been fetched, it is then passed to the Decode stage which also generates the local address for the stack cache. The third and final stage of the JOP pipeline is the Execute
Stack stage. The last stage contains the circuitry to perform operations on the stack, and also contains what is normally considered the Write Back stage in a microprocessor.

The JOP is situated as a series of Very high speed integrated circuit Hardware Description Language (VHDL) files that get compiled to a *.bit* file by Xilinx ISE and then loaded onto the FPGA via a Joint Test Action Group (JTAG) interface using Xilinx iMPACT. Separately to the compilation of the JOP architecture, the Java program is compiled by the Java Development Kit (JDK) provided by Sun microsystems into the bytecode instructions of the JVM.



hex		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 10. SBox[2]

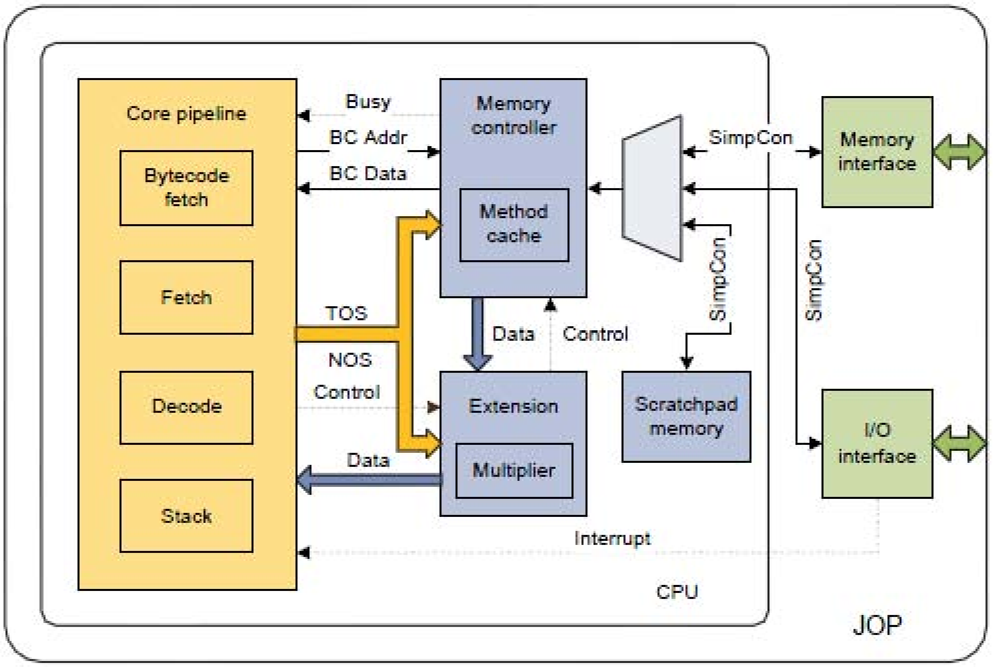


Figure 11. JOP Block Diagram

III. Methodology

The methodology chapter first explains the proposed countermeasures to protect the JOP running the two cryptography algorithms. Then this chapter covers the overview of the set up of the experiments. Finally, an explanation of the experimental test setup and description of experimental test procedures is provided.

3.1 Proposed Countermeasures against RSA

In the protection of circuits running RSA cryptography, protection methods are based off the knowledge that processor mathematical operations have a unique power signature [4]. The proposed hypothesis is that through the addition of duplicate hardware that have the same functionality but are structurally different, the correlation between the processed data and the EM emissions can be significantly reduced. Where normal methods to add EM ”noise” to systems overlay the noise over the ”useful” EM information, this thesis explores the idea that the EM ”noise” created not only changes the EM emissions, but also completely removes the previous ”useful information” dynamically changing which hardware is used. The two methods used to test this hypothesis are to have two functional units of differing signatures to be separated in time (alternate use) or at the same time but separated with data (both execute at the same time, one is sent the correct data and one is sent the incorrect data). To test the time differential nature of separate functional units, the add operation was modified to have two different architectures, a Ripple Carry Adder (RCA) and a Carry Look Ahead (CLA), and the JOP alternates which adder is used each time an addition operation was executed (referred to as the Double Adder). The RCA uses an even distribution of power to add two numbers due to its many cascaded stages, while the CLA uses a front loaded distribution of power to add two numbers due to its look-ahead nature. Using these two add operations, regardless of the data being added, a given add operation will either be front loaded or evenly distributed in power, and the subsequent add operation will have the other distribution of power. To test the data differential nature of separate functional units, the multiply operation was modified to have two "asynchronous" Booth multipliers, referred to as the DAILMOM, where one multiplier was supplied the correct operands while the other multiplier was fed arbitrary data. The arbitrary data being fed to the multiplier was selected to give the multiplication operation a front loaded appearance, a back loaded appearance, split on either side in time, or a middle loaded appearance. This loading factor changes for each multiplication operation

3.1.1 Dual "Asynchronous" In-Line Multiplexed Out/inputs Multiplier (DAILMOM).

The basic JOP before any modification uses an external synchronous multiply unit. The original JOP multiply unit is shown in Figure 12, it requires 32-bits for each of the two inputs that are fed directly into the multiplier, then the multiplier is clocked until the number of clock cycles necessary to multiply the two numbers is complete. The resulting 64-bit value is returned to the program (although in actuality, only the lower 32-bits is returned, to get the higher 32-bits requires a long int multiplication).

To make the multiplier unit more hardened against SCA, the multiplier that came with the fresh install of the JOP was replaced (over-ridden) with the DAILMOM. First, the basic underlying structure of the multiplier was changed from a synchronous multiplier to an "asynchronous" Booth multiplier. The term "asynchronous" is being used here loosely to describe that there is not clock signal to control the flow inside the Booth multiplier, so the multiplier uses purely combinational logic. Then, the multiplier unit is doubled so that there is the circuitry for two multipliers, basically

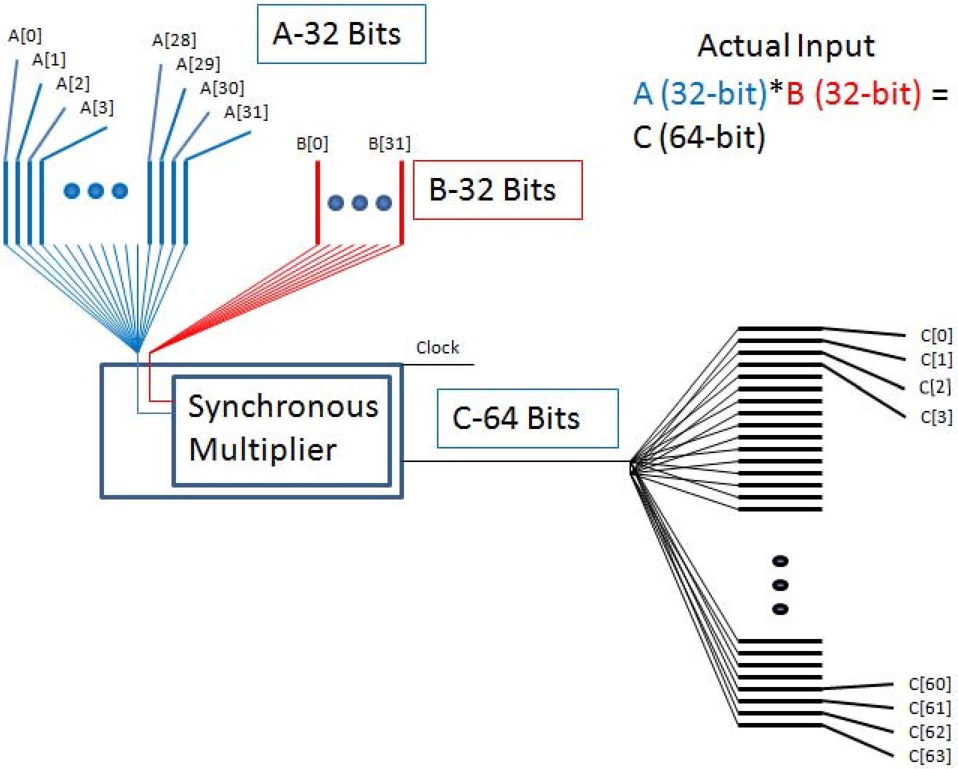


Figure 12. Original synchronous JOP hardware multiply unit

allowing for two 32-bit numbers to be multiplied simultaneously. One of the multipliers is used for the actual input values of A and B, while the other multiplier acts to confound the power signature. The second multiply unit is the exact same, and runs in parallel with the multiply unit that is actually producing the value. A feature of this design is that the user of this multiply unit can manipulate the shape of the power signature of a given multiply operation by changing the values of A2 and B2. Furthermore to confound the power signature of values being sent and received from the multiply unit, all values are interleaved with the corresponding confounding value. For example, A1 (real input) and A2 (confounding input) are interleaved together such that the first four bits of the first byte representing the input passed to the multiply unit is $A1_1$,$A2_1$,$A1_2$,$A2_2$,$A1_3$,$A2_3$,$A1_4$,$A2_4$, thus two 32-bit numbers interleaved together makes one 64-bit number. The same is done to B1 and B2, and the two outputs C1 (actual output) and C2 (confounded output) are also interleaved when being passed back to the JOP. Figure 13 better illustrates this new multiply unit. With the introduction of the random values being processed and the multiplexed signals, the data coming off the multiply unit no longer is directly correlated with the data, thus helping to obfuscate the power signature (see the results in Section 4.3.3). Furthermore, because the “noise” during the multiplication is also created by a multiplier and is not generated white noise, it is nearly impossible to separate the two signals because there is no overt distinction. Lastly, de-coupling the multiply unit from the clock makes correlating traces much more difficult.

The DAILMOM architecture is made up of several parts in VHDL. The first part is the structural definition of a Booth multiplier, with 32 cascaded stages in each multiply unit. Since the DAILMOM VHDL description does not include a clock, or more specifically it uses only combinational logic, it needs to contain control logic so that it knows when to multiply and it can synchronize with the rest of the JOP when the result is presented on the DAILMOM output port. The entire VHDL definition for the DAILMOM can be seen in Appendix B, Listing B.1.

3.1.1.1 Dual Adder.

The Dual Adder is similar to the DAILMOM, except that the purpose of the Dual Adder is to obfuscate the addition operation by making every two consecutive add operations look different. This obfuscation via the Dual Adder is accomplished by having two adders that the JOP alternates between each time an add operation is evaluated. The two adders, the CLA adder and a RCA, are written in VHDL with a central driver unit. The central driver unit is added to the JOP in such a way to overload the original add operation, such that the Dual Adder driver is called for each add operation instead of the previous addition module. The carry look ahead adder VHDL can be seen in Appendix B, Listing B.5. The ripple carry adder is similar in port definition as the carry look ahead adder, but its very different than the carry look ahead in implementation. The ripple carry adder also uses two other pieces of hardware, the full adder and a carry buffer (the carry buffer functions to keep the stages from being optimized out by Xilinx). The architecture of the ripple carry adder can be seen in Appendix B, where Listing B.2 shows the high level RCA definition, Listing B.3 shows the full adder definition, and Listing B.4 shows the carry buffer.

3.1.2 Overloading the original JOP multiplier and adder.

After the code for the multiplier and adder was made and tested, it had to be added to the JOP in such a way that the JOP would use the double adder when a "+" was encountered in the Java code, and the DAILMOM would be used whenever a "*" was encountered in the Java code.

3.1.2.1 Connection to SCIO Bus.

To connect these new math hardware units to the JOP hardware was done through the SimpCon Input/Output (SCIO) bus. A central "math" driver was created and connected to the SCIO bus. This central driver allowed for one central point for the JVM to go to when performing math operations, and controlled the connections to both the adders and the DAILMOM. The code for the math driver can be seen in Appendix B, Listing B.6.

3.1.2.2 Software Interaction with hardware.

Access to the math driver and any hardware in the JOP was made by defining the ports in the const.java file. These are the ports that are read or written to by Java when accessing the math driver. These port definitions can be seen in Appendix B, Listing B.9. With the ports defined, hardware can be used with a simple Native.wr() or Native.rd() to write to or read from a port.

3.1.2.3 Overriding Base Multiplier and Adder.

To get the JOP to use the hardware instantiated adders instead of the default add operation for the JOP, the assembler instruction in the JOP had to be overwritten to tell the JOP to look for a software implementation of the add operation. This was done in Jvm.asm, and the code pertaining to the integer multiply and add operation respectively can be seen in Appendix B, Listing B.7.

Without a definition for *imul* and *iadd* in jvm.asm, the compiler looks instead to jvm.java for a software instantiation of the integer addition. The code inserted into jvm.java for the adder and new multiplier can be seen in Appendix B, Listing B.8. The code for the double adder is set up to alternate which adder is used, and when an adder is used, the first value is loaded into the first add buffer then the second value is loaded into the second add buffer and the result is read. The code for the multiplier is set to select a confounding input, interleave the two input values, wait for a completion bit, and then un-interleave the output.

Now, with the multiply and add operation tied to the hardware object, when an integer multiplication or addition is called, say, 8+9, 8 and 9 will be loaded into one of the add module input ports and the result, 17, will be read from the same adder module output port.

3.1.3 RSA Code.

To run RSA code, all that is required is a method to store large integers (at least greater than the 32 bit limit of the JOP) and perform efficient power and mod functions on the numbers. Originally for this thesis, the provided functionality from Java for large numbers was provided as part of the BigInteger java class. This class was not able to be implemented on the JOP because the JOP is a very small version of the JVM environment and does not have the support of the full JDK package. Because the BigInteger class could not be run on the JOP, the first idea was to write a new version of the power mod function from scratch, implementing a simple binary exponentiation. After a short time it was decided that this was beyond the scope of the thesis so the use of the BigInteger class on the JOP was re-evaluated, and eventually an abridged version of the BigInteger class was created that could run on the JOP and had the needed functionality of RSA. Another difficulty in doing this was that BigInteger had data dependencies in the MutableBigInteger class and the SignedMutableBigInteger class that had to also be disregarded that would not be able to run to allow them to be added to the JOP. In the end, the final version of the BigInteger class had lost its ability to securely create prime numbers, the ability to test the security of them, and several string functions that were also unnecessary.

However, the modified BigInteger was able to create a BigInteger number from a byte array, perform the ModPow() function on the number, and then pass the finished result as a byte array back to the calling function. The functionality of prime number creation would have been nice, but was totally unnecessary because for testing, the same prime numbers were always used and there was no need for generation of new keys.

3.2 Proposed Countermeasures against AES

In the protection of circuits running AES cryptography, protection methods are based off the knowledge that the memory operations are the main avenue for information leakage. With this in mind, the proposed hypothesis is that through the addition of memory units, the correlation between the processed data and the EM emissions can be significantly reduced. This reduction in correlation is because the values being stored are split into two values, and each of the resulting values will always have a HW of 16 regardless of what the original value was. The equal HW is important because the leakage model for CMOS circuits use the HW of the plain texts to correlate them with the actual values. The two methods used to test this hypothesis are to have doubled variable cache in the execute stage to obfuscate the storage of data during the execute stage and the other method is to have a doubled heap space to obfuscate the storage of variables in the heap. In both cases, the obfuscation happens by splitting the current value into two pieces, all the even bits and all the odd bits, and then storing all the even bits with their inverse as one value and storing all the odd bits with their inverses as a second value. Having the bits and their inverse in the same value, the Hamming Weight of every value being saved is always equal to half the bit length. As can be seen in the code for the RAM segment in Figure 14, ram_dout_buffer is the signal name given to the information that would normally be saved in the heap,

Table 5. JOP execution Example A=B+C*D

Stack	JVM
push B	iload_1
push C	iload_2
push D	iload_3
*	imul
+	iadd
pop A	istore_0

and it is being split between the even bits in sram_data_out_0 and the odd bits in sram_data_out_1, along with the inverted bits. Then, when reading a value from the RAM, the two values that get read are sram_data_in_0 and sram_data_in_1 as can also be seen in Figure 14, they get combined to re-form the actual data.

3.2.1 Implementation of Double Stack.

The JOP, as previously mentioned, uses a stack based architecture. In the JOP, when an addition command like "iadd" is encountered, the JOP adds the top two elements in the stack and then saves the result back to the top of the stack. For example, Table 5 shows an example JOP stack instruction.

The Double RAM's goal to obfuscate EM emissions is to change the way that values are saved during the actual execution of the code. Using the two new confounded values for each value that would be pushed onto or pulled from the stack, the Double Stack only stores values with a HW of 16. These two values are also stored simultaneously onto the on-chip "RAM." The full code for double stack is presented in Appendix D.

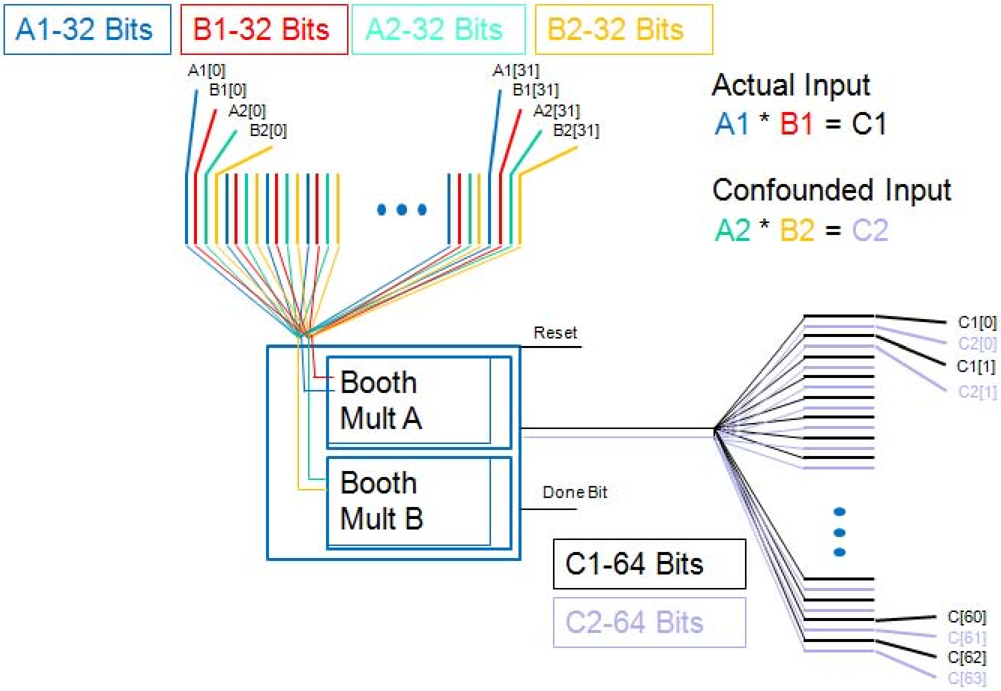
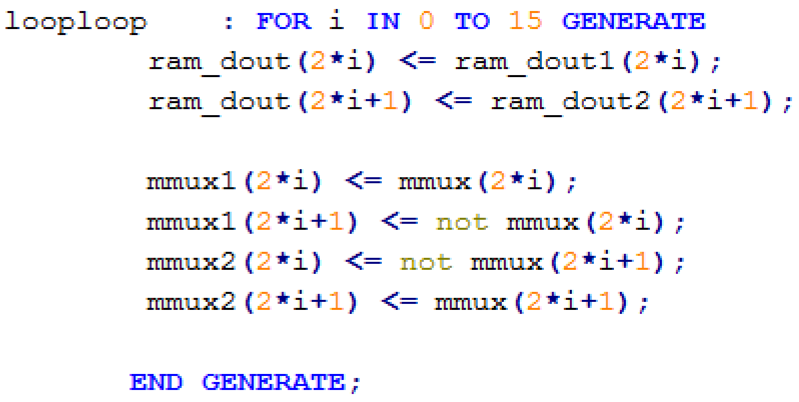


Figure 13. Dual Asynchronous In-Line Multiplexed Out/inputs Multiplier (DAIL-MOM)



```
loop loop     : FOR i IN 0 TO 15 GENERATE
          ram_dout(2*i) <= ram_dout1(2*i);
          ram_dout(2*i+1) <= ram_dout2(2*i+1);

          mmux1(2*i) <= mmux(2*i);
          mmux1(2*i+1) <= not mmux(2*i);
          mmux2(2*i) <= not mmux(2*i+1);
          mmux2(2*i+1) <= mmux(2*i+1);

         END GENERATE;
```

Figure 14. VHDL to interleave signals

3.2.2 Implementation of Double RAM.

When the program is first downloaded to the FPGA board, it is saved into the off-chip RAM. Because the program is normally only a few kilobytes, and there is one megabyte of RAM, to implement the double RAM the current ram space was cut in half. Every time a value was written to or read from RAM, one of the split values was written to the low address space and the other corresponding value was written into the high value space. Previously, the RAM module was called to read or write a value, it would read once or write once. With the redesigned however, when the RAM was called to read or write one value, two values consecutively are read or written. The previous model of the RAM interface took 5 clock cycles to read and 6 to write, the modified version of the RAM then took 12 clock cycles to read and 16 to write, effectively doubling the time for RAM actions. The code for the modified RAM module can be seen in Appendix E.

3.2.3 AES Code.

The original AES code worked on the JOP, but it required a some major modifications to greatly reduce the necessary Java garbage collection and to reduce the number of objects that needed to be swapped in and out of memory. The original encryption program uses Sbox.java to function as a look-up Table for the sboxes, it uses KeyExpansion.java to find the keys that get XORed at the various stages and then it uses AES_Main.java as the main driving unit for the encryption. To reduce the overhead of object swapping, all of the code was put into RSA_Main_Independant.java with two functions, the main function which does almost everything in AES and the shifting function which does an important shifting operation as part of the Mix-Columns. Originally this was not possible, but by reworking a lot of the AES code and optimizing everything, the main function absolutely maxes the size of any function able to run on JOP to the extent that adding even one more line of code causes the function to throw a “function too big” error when run on the JOP. The Code is included as Appendix C.

3.3 Testing Overview

The testing overview section of this chapter covers the overview of the set up of the experiments by outlining the goals and hypothesis, approach, system boundaries, system services, workload, performance metrics, system parameters, factors, evaluation technique and the experimental design.

3.3.1 Goals and Hypothesis.

With the susceptibility of microchips to SCA attacks, the first goal of this thesis is to establish a security baseline. This baseline is the result of running cryptographic code on the JOP implemented on the ML506 Virtex 5 evaluation board without any countermeasures employed. Once the baseline is establish, the next goal of this thesis is to implement each proposed countermeasures individually and compare the resulting security against the established baseline.

The hypothesis of this thesis is that through the use of a dynamic hardware set, that the security of cryptographic code running on the microprocessor can be made more secure without changing the software, with respective area and time costs. The two proposed hardware sets to increase the security of RSA are the Double Adder and the DAILMOM, while the two proposed hardware sets to increase the security of AES are the Double Stack and the Double RAM.

3.3.2 Approach.

The goal to establish a baseline takes several steps to accomplish. First, the cryptographic code needs to be enabled to run on the JOP; for example, the stack size of 128Mb has to be increased to 256Mb to prevent a stack overflow due to the computational need in the case of RSA. Once the code is running, Riscure Inspector is used in conjunction with the Riscure XY table to locate the best hot-spot on the FPGA. This hot-spot is needed in order to know where to acquire the best traces for analysis. At this point, the power traces will be recorded for SCA attacks.

Establishing the security of the proposed countermeasures use a similar approach. Once the proposed countermeasures have been implemented on the JOP, a hot spot for the new JOP is located, traces recorded and then the cryptosystems are analyzed again. The information from compromising the JOP with the countermeasures are compared against the information from compromising the JOP without countermeasures.

3.3.3 System Boundaries.

The System Under Test (SUT) is the JOP, which includes the architecture needed to implement the JOP on the FPGA. The actual test values are the recorded EM emissions of the JOP and its architecture during testing. Furthermore, the JOP is the only hardware implementation tested during this research. In addition to the JOP being the SUT, the Components Under Test (CUT) are the basic JOP architecture, and the specific hardened architectures of the DAILMOM, the Dual Adder, and the Double Stack and the Double RAM. Figure 15 shows a block diagram of the SUT.

3.3.4 System Services.

The system services of the JOP are a fully functional, pipelined microcontroller with the capability to run either the RSA or the AES cryptography algorithm. These are chosen because they are real world security algorithms, and thus the results of this research has a real world relevance.

3.3.5 Workload.

The workload of the AES encryption and RSA encryption is the plain text that is sent to the JOP via a laptop over a serial cable.

3.3.6 Performance Metrics.

The performance metrics are tailored to best measuring the increased security provided by the components under test. First, tests are made to establish a baseline with which to compare the rest of the data. As previously mentioned, the initial baseline is completed by compromising the JOP without any additional hardware, and then the new hardware is implemented and tested for security. In the case of AES, the performance metric is the minimal number of traces needed to find the key with a 95% confidence. In the case of RSA, because the Inspector software is not set up to find the key of the optimized version of bit-group exponentiation used by Java, the performance metric is the visual analysis using SPA.

These performance metrics are chosen because they directly reflect the real world security of the encryption algorithms.

3.3.7 System Parameters.

The system parameters for the given system reflect the characteristics of the system that if changed will have an effect on the responses. The three system parameters are the FPGA evaluation board used, the probe used, and lastly the oscilloscope used.

The most influential system parameter is the FPGA evaluation board that is used because each evaluation board has a different Virtex 5 FPGA which changes the EM signature. This is because the architecture of the FPGA strongly influences what the EM off the FPGA will look like given a cryptosystem. The three options for FPGA evaluations boards are the ML505, the ML506 and the ML507. For this research, after the ML505 board was damaged beyond repair, the ML506 was used with the Virtex 5 XC5VSX50T.

The next most important system parameter is probe used to measure the EM traces. The Wilcox RF probe and EM probes were originally used, but then replaced by the Riscure EM probe because the Riscure EM probe has the best resolution for the recorded traces.

The final system parameter was the oscilloscope that was used. Provided that the oscilloscope was able to provide the necessary functionality of a fast enough sample rate, the oscilloscope used was not nearly as important as the other two parameters. Although the Lecroy 7zi WaveRunner was initially used, the Lecroy 8zi WaveMaster was used for all the results in presented in this research.

3.3.8 Factors.

The factors of the system are based on the system parameters. The factors for the target FPGA include the FPGA itself and its surrounding hardware, the interface used, and the power supply.

The oscilloscope had several different factors to choose between. The first and most important factor is the sampling frequency. Based on a desire to capture all of the information every 99Mhz clock cycle, a frequency above 200Mhz has to be chosen. The Nyquist frequency is twice the highest frequency that needs to be captured and is the slowest sampling rate needed to capture relevant information, thus 198 Mhz is the Nyquist frequency for the 99Mhz clock. When possible, data is taken at 1Ghz to provide a fast enough sampling rate and enough data points to capture the important portion of the encryption algorithm. The next factor is the use of a physical filter to remove unwanted frequencies. If a filter is not used, harmonics higher than half the sample frequency would be aliased. To avoid the aliasing at 1Gs/s for frequencies higher than 500Mhz, a low pass filter with a 200Mhz cut-off is selected. The final factor was the use of coupling, where DC coupling allows all of the signals to pass to the sensor, and AC coupling passes only the AC components of signals while attempting to remove a DC bias. DC coupling is used in order to allow the largest amount of recorded data.

3.3.9 Evaluation Technique.

The evaluation of different protection techniques are assessed using actual, measured results. For this research, actual results are better than simulated results because the actual results directly represent the real world threat. Data is collected using a Lecroy oscilloscope and Riscure EM probe, and analysis is done using the Riscure Inspector analysis program on the computer.

3.3.10 Experimental Design.

Experiments are split up into AES tests and RSA tests because the counter measures employed to protect AES do not affect RSA vulnerabilities and likewise RSA SCA countermeasures do not affect AES security. When testing AES, ten data points are be taken at three different locations identified as leaking the most information while employing each of the three hardware sets (the hardware without counter measures, the hardware with the double stack, and then the hardware with the double

RAM). Then each position is compared within itself for outliers. Once each position contains only the data points within a 95% confidence, all of the data points for each of the hardened architectures is compared against those from the baseline using a 2-sampled t-test.

In testing RSA, the actual key cannot be derived from the data due to an incongruence between Java's optimized bitgroup exponentiation and the regular bitgroup exponentiation expected by Inspector. Thus, it is shown how Inspector would normally find the key, and show visually the correlation of the traces, and look for obvious differences in the correlation as proof of an increase in security.

3.4 Testing Setup

The testing equipment for all experiments used a Lecroy oscilloscope, the Riscure XY table, the Riscure EM probe, the FPGA board and related peripherals and a laptop running Riscure Inspector. The Lecroy oscilloscope used was an 8zi WaveMaster or the Lecroy 7zi Waverunner oscilloscope depending on availability. Due to a limit with Riscure Inspector, the largest amount of memory available was 20 MegaSamples regardless of the oscilloscope, and the max sample rate was 10 GigaSamples per second. See Figure 16 for an example of the 7zi model.

Initial tests for cryptographic security were done using the Wilcox EM Probe and the Wilcox RF probe. See Figures 17 and 18 respectively. During those tests, When the EM probe was used, data was taken from the underside of the FPGA board. When the RF probe was used, data was taken by touching the probe to the contact for T18, the power capacitor.

Due to poor results with these probes on the previous ML505 board, the RISCure XY table with the RISCure EM probe were used instead (see Figure 19).

The final decided upon set up can be seen in Figure 20. As is seen in the pic-

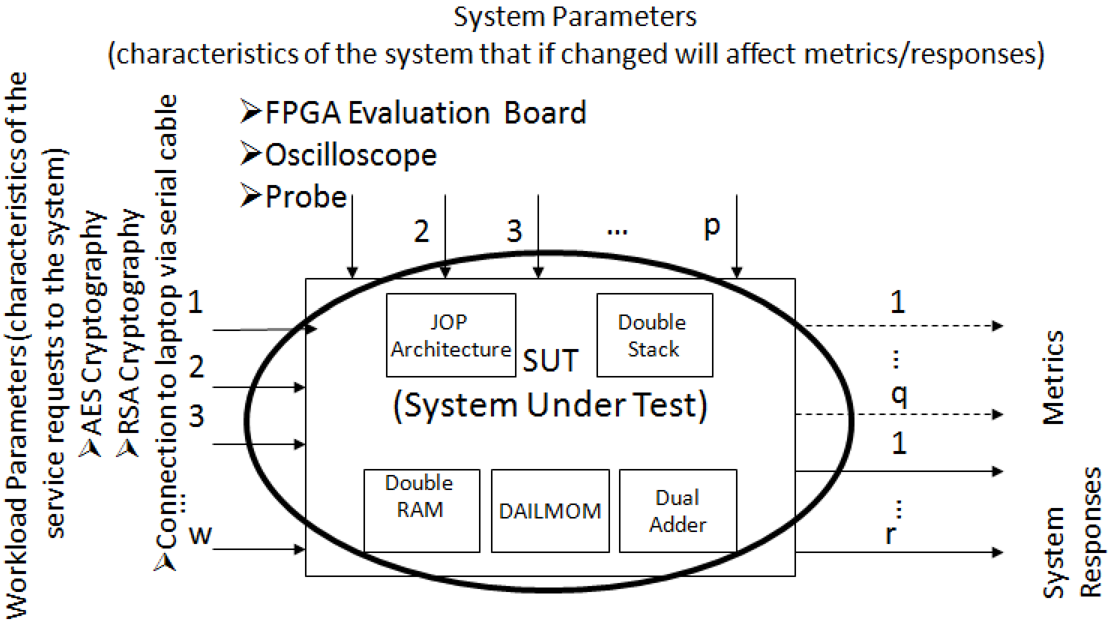


Figure 15. System Under Test (SUT)

Figure 16. Lecroy 7zi oscilloscope

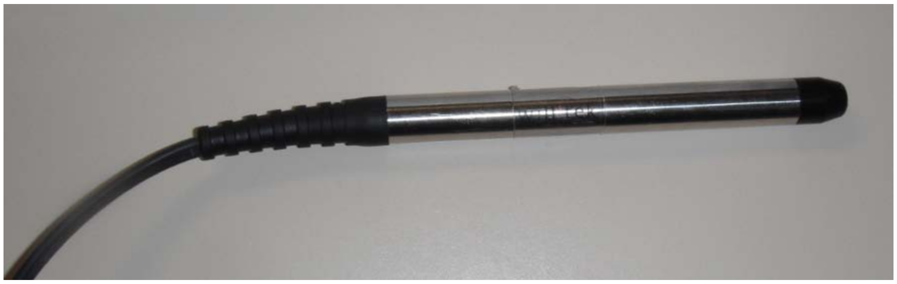


Figure 17. EM Probe

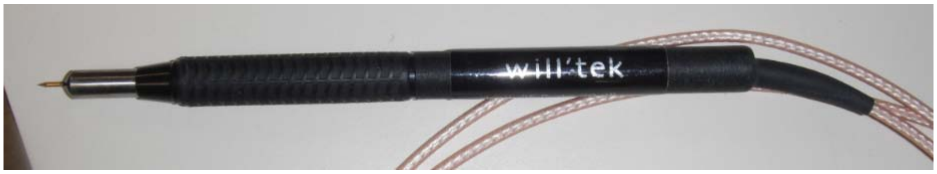


Figure 18. RF Probe

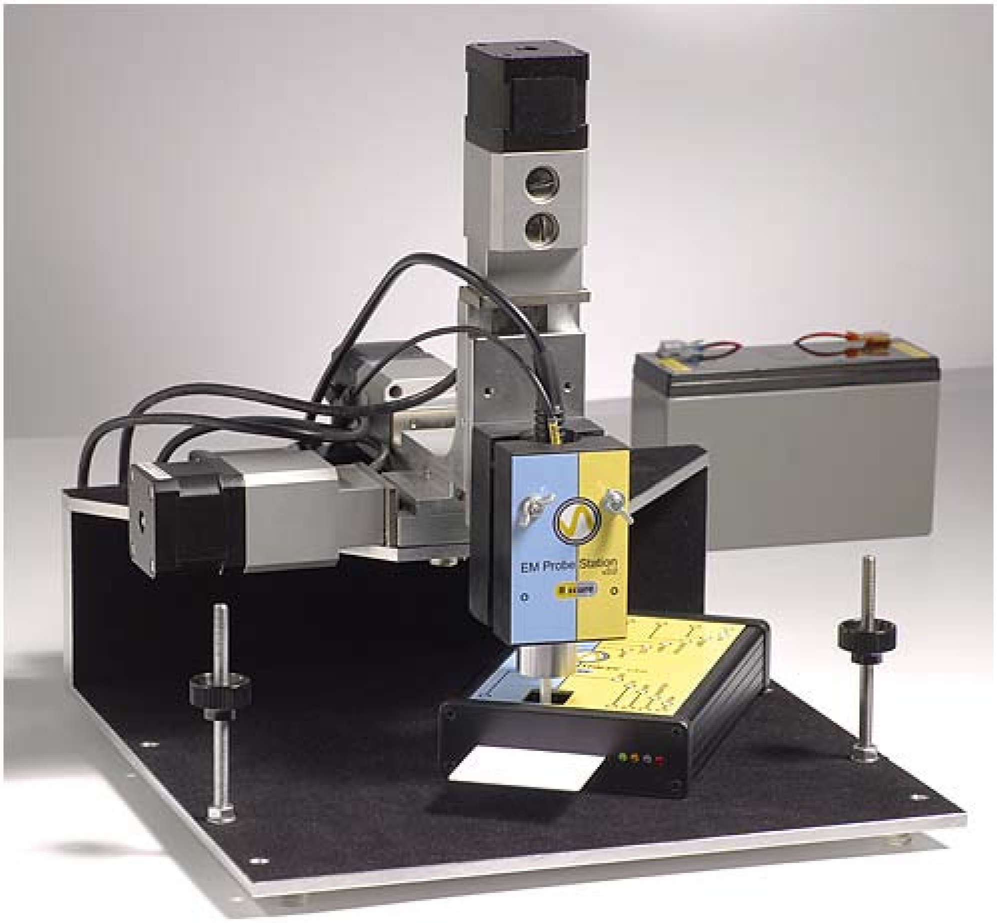


Figure 19. RISCure XY table with EM probe

ture, the FPGA board has the connections of the provided power supply, the JTAG interface which was used to program the FPGA, the serial cable with which the Java instructions were sent to the JOP, and the trigger attached to two of the pins on the pin jumper. The computer had the other end of the JTAG interface, the serial cable, and it had the key dongle for Inspector to allow Inspector to be used on the laptop. The oscilloscope, seen in Figure 21 shows the output of the EM probe connected to the second channel, and the trigger connected to the third channel. Lastly was the power supply, seen in Figure 22, supplied power to the active Riscure EM probe.

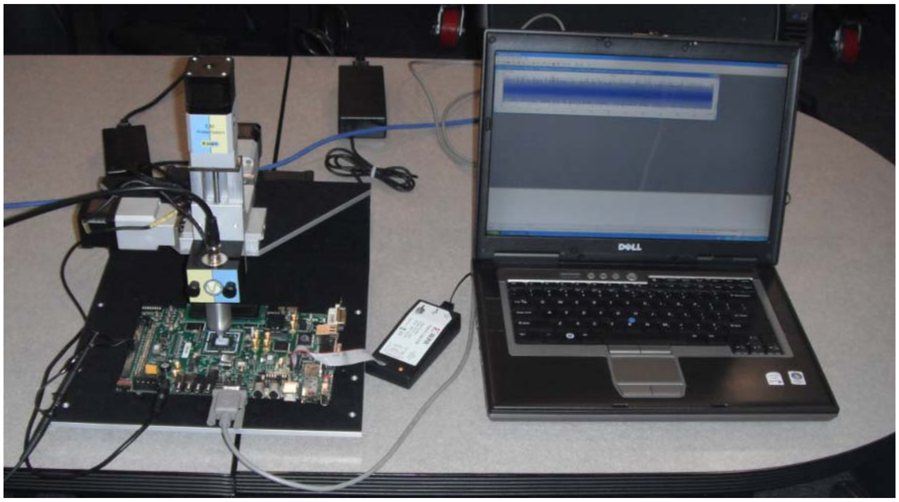


Figure 20. Laptop and Riscure XY Table

Figure 21. Lecroy oscilloscope being used

Figure 22. Power supply to Active EM Probe

IV. Experimental Results

Chapter 4 contains the experimental results and is split into three sections. The first section is labeled "Initial Data Collection" and outlines the original flow of experiments, initial hang ups, layout considerations on the Virtex 5, and then some code considerations. The second section outlines the AES attacks on the JOP and the results of the Double Stack and the Double RAM. The final section of Chapter 4 outlines the attacks on RSA and the results from the Double Adder and the DAILMOM.

4.1 Initial Data Collection

4.1.1 Initial Attacks.

To begin the thesis experiments, a baseline was needed to juxtapose the results from the proposed circuitry countermeasures to show increases in security. This baseline consisted of the basic JOP using only the original circuitry.

The ML506 was chosen to be the physical test platform for this research because there were a least two of ML506 boards available (for redundancy) and because it did not contain a PowerPC core (which would have been wasted space). Although the ML506 has additional DSPs that were not necessary, they did not affect the usable area on the FPGA the way the PowerPC does. Using the new ML506 evaluation board, and the new Inspector EM probe, the trace seen in Figure 23 resulted. This trace clearly shows all 10 rounds of the AES encryption.

To try to pinpoint the best location to attack the AES encryption, an XY plot was made of all the data and then the 99Mhz frequency was pinpointed for the traces doing the full AES encryption. See Figure 24 for the XY plot of the 99 Mhz signal during the full AES encryption.

Understanding the vulnerability of the S-box substitution in AES encryption, the probe was set up to trigger during the first SubBytes function. This resulted in the XY plot shown in Figure **??**. From this Figure, and looking at the traces that resulted, trace 44 (shown in Figure 25) was chosen as the point over which to position the probe due to it having the clearest signal and showing the largest number of peaks in the trace per substitution. The position on the FPGA of trace 44 can be seen in Figure 24 as the small light green box near the middle of the chip with a white boarder around it. The trace chosen appeared to show all 16 Bytes being substituted well, while some of the other traces had overall stronger 99Mhz signals, they did not show all 16 bytes as distinctly.

After finding a good location with a clean representative trace showing all 16 byte substitutions, the probe position was set and a data capture was done capturing 20,000 traces resulting from random inputs. This was done at 1 GS/s, with 1Ms of data and a 200 Mhz filter. The average trace of the 20,000 traces is shown in Figure 26.

The straight data without processing was analyzed over the course of 14 hours to verify Inspector's ability to derive the correct key on a desktop computer using Windows 7 x64, interfacing with an Nvida GTX460 video card with 336 Compute Unified Device Architecture (CUDA) cores. The correct key was found, verifying the data collected did contain the necessary key information. After verifying this result,

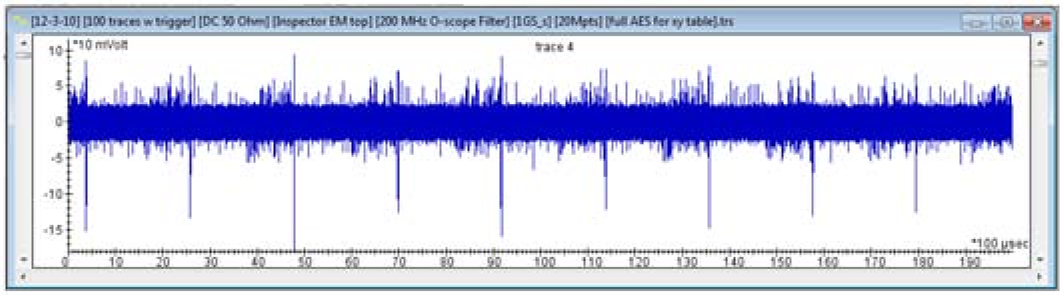
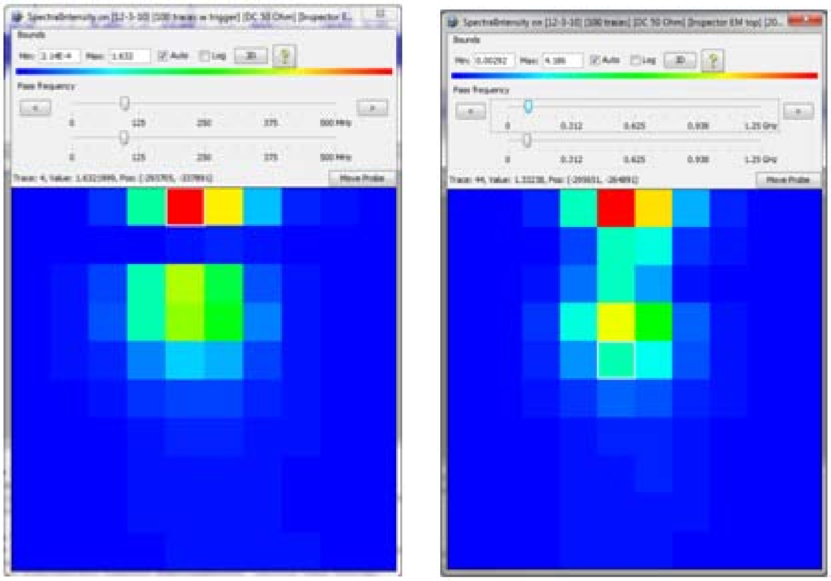

[12-3-10] [100 traces w trigger] [DC 50 Ohm] [Inspector EM top] [200 MHz O-scope Filter] [1GS_s] [20Mpts] [Full AES for xy table].trs
*10 mVolt
trace 4
10
5
0
-5
-10
-15
*100 µsec
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190

Figure 23. Example traces showing periodic nature of AES (10 rounds)



(a) Full AES encryption

(b) S-box substitution

Figure 24. Plot of the surface of the FPGA during the full AES encryption (a) and during the first S-box substitution (b) showing only the 99Mhz signal where red shows the largest amount of signal and blue shows the least amount of signal

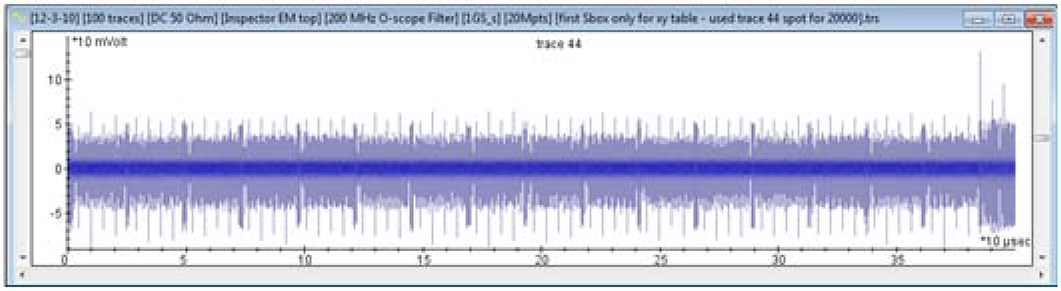


Figure 25. Trace chosen of S-box substitution

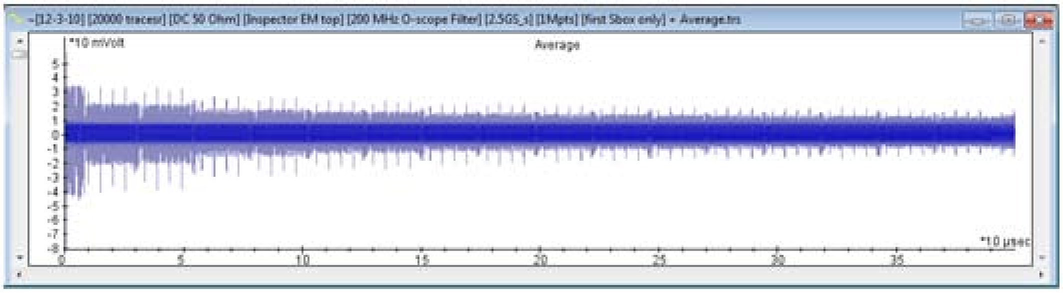


Figure 26. Average S-box substitution trace

different data processing techniques were used to increase the signal to noise ratio and reduce the needed traces down from 20,000 and time down to several seconds. To do this, the processes of trace alignment, absolute values, low pass filter, and synchronous resampling at 99Mhz clock frequency. Of the options explored, taking the absolute values of the traces and synchronous resampling the traces at the clock frequency had the greatest effect. Taking the absolute values of the traces had a large effect because in the analysis of the EM spectrum, the orientation (negative or positive) of the peaks does not matter. The resampling at the clock frequency means that the values surrounding each clock tick was replaced with a single value for that clock cycle. Although some information was lost by doing this, time savings for processing was dropped to less than 1% of the previously needed time.

After the preprocessing was complete, it was found that for the given traces, the minimal number of traces needed was 80 to produce the correct key guess.

4.1.2 Evaluation of More Dense Mapping Techniques.

Although the correct key can be found by looking at the SubBytes fuction on the JOP in its current form, it was thought that through better mapping onto the FPGA better results could be attained. The JOP was currently mapped onto the FPGA through a non-deterministic random manner producing the spread out pattern seen in Figure 27. Due to its random placement, it was thought that through reducing the footprint of the JOP using Xilinx PlanAhead and forcing the JOP to be mapped onto a smaller area in the middle of the FPGA, the signal to noise ratio would be higher rendering the JOP more susceptible to SCA attacks and thus reducing the number of possible confounding variables. The resulting spectral intensity plot looked exactly the same as plot before changing the mapping of the JOP, as can be seen in Figure 29. Knowing this, and through several more tests on different random arrangements of the JOP on the FPGA showed that the largest intensity of the power traces was always from the same location, centered over position 44 (seen in Figure **??** as the small light green box near the middle of the chip with a white boarder around it).

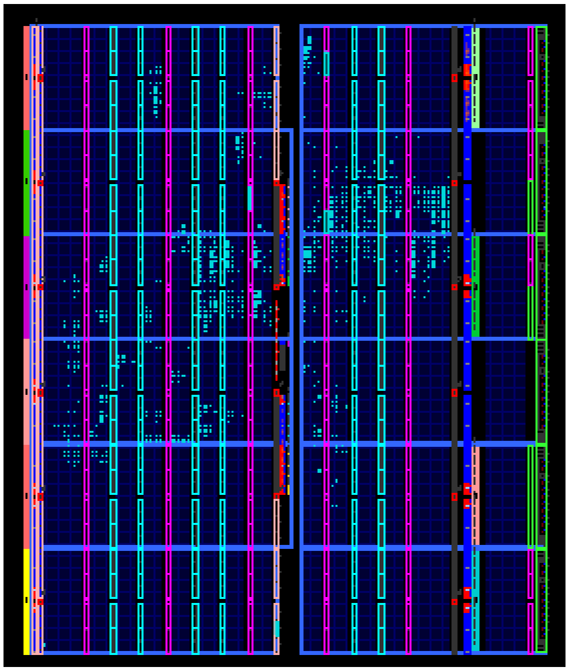


Figure 27. Original JOP architecture placed on FPGA

Upon further investigation, it appeared that the coupling capacitors under the FPGA were causing the strong information leakage in the middle of the board, and when the probe was centered over capacitor C1 which connected $FPGA\_AVDDD$ to $GNDA\_FPGA$, regardless of the mapping of the JOP on the FPGA, the signals were always the best. See Figure 30 to see the capacitor with the most clear leakage signal.

Knowing the importance of the capacitor bank in the leakage of the information, it was thought that perhaps better results could be acquired by looking with the probe underneath the board. However, after attempting to record the 16 cycles of the SubBytes command under the board resulted in Figure 31, which was significantly less clear than Figure 25 as seen by taking the same data reading from the top.

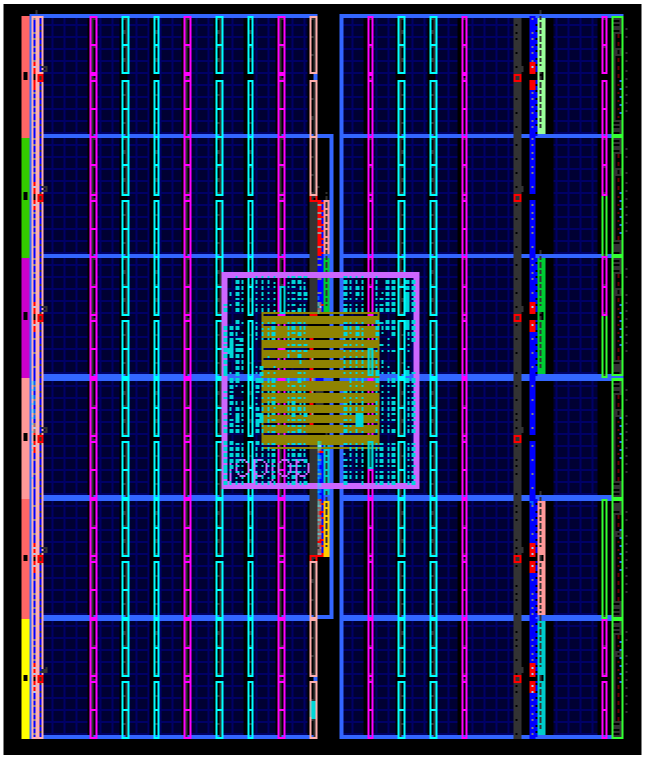
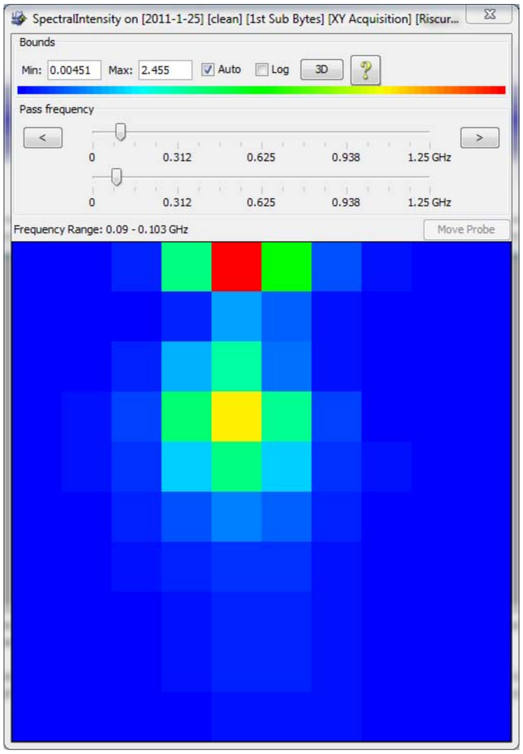


Figure 28. Consolidated JOP architecture placed on FPGA


SpectralIntensity on [2011-1-25] [clean] [1st Sub Bytes] [XY Acquisition] [Riscur...

Bounds

Min: 0.00451 Max: 2.455 ☑ Auto ☐ Log

Pass frequency

Frequency Range: 0.09 - 0.103 GHz

Figure 29. Spectral Intensity Plot for Dense JOP

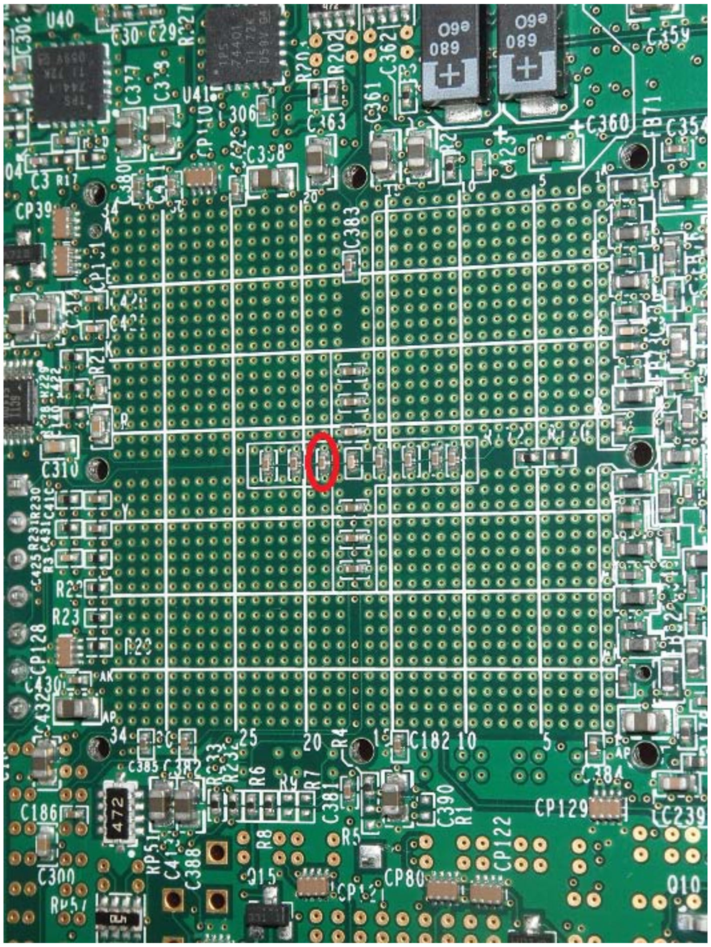


Figure 30. Power capacitor giving best SCA information leakage circled in red

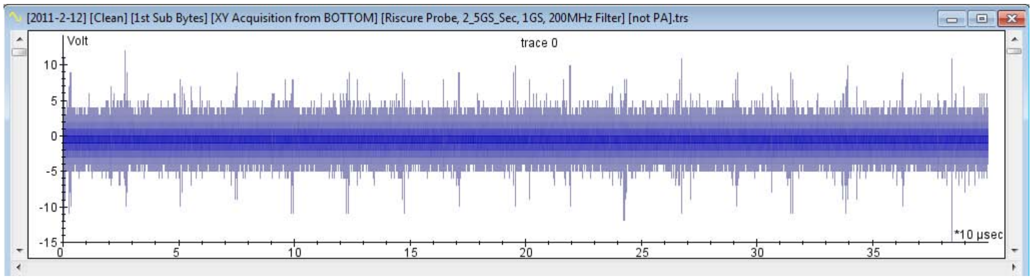

[2011-2-12] [Clean] [1st Sub Bytes] [XY Acquisition from BOTTOM] [Riscure Probe, 2_5GS_Sec, 1GS, 200MHz Filter] [not PA].trs
Volt
trace 0
10
5
0
-5
-10
-15
0
5
10
15
20
25
30
35
*10 µsec

Figure 31. AES SubBytes as seen from the bottom of the FPGA

Table 6. FPGA Design Resource Utilization

	clean	ram	timing	iadd	imul	ALL
Register	4	4	4	4	11	12
LUT	8	8	8	8	20	21
Block Memory	3	3	3	3	3	3
Clock Manager	5	5	5	5	5	5
Global Clock Buffer	3	3	3	3	6	6
IO	14	14	14	18	18	18

Furthermore, it was interesting to note the difference in area taken up by the "clean" version of the JOP on the FPGA and the area taken up by different obfuscation techniques. The area taken up by the original JOP on the FPGA and the corresponding areas for the proposed countermeasures can be seen in Table 6.

Although most of the obfuscation techniques required only a negligible amount of resources, the DAILMOM was resource heavy. Due to the large disparage between the clean JOP and the JOP with the DAILMOM, there was a risk that it might be easier to extract the key from the clean JOP just because there was less circuitry, not because of anything in specific the circuitry was doing. Thus, to rule out this confounding variable, all tests were done with "ALL" hardware instantiated on the JOP, although the hardware was not always enabled. For example, when running tests with the clean JOP without countermeasures, the circuitry for both the double adder and the DAILMOM were present, although neither the iadd operation nor the imul operation was overridden.

Lastly it was also important to note the proposed time increases for the actual time increases for the countermeasures, as can be seen in Table 7. The "Actual Time Increases" are the actual increase of time for the JOP to perform the encryption algorithm while the "Proposed Time Increases" account for what the time increases would be on a JOP where the countermeasures had been fully integrated. Being "fully

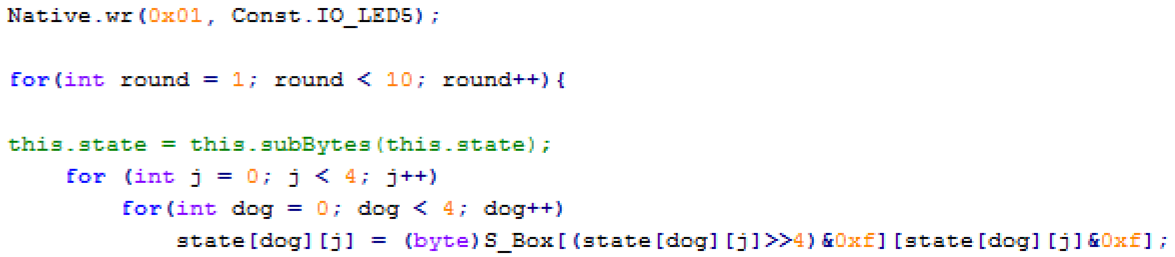
Table 7. **FPGA Design Resource Utilization**

	Proposed Time Increase	**Actual Time Increase**
Double Adder	0%	150%
Double Multiplier	0%	2000%
Double Stack	0%	0%
Double RAM	250%	250%

integrated" denotes that the JOP math operations do not need to be overridden, instead they are the only method the JOP has for addition and multiplication and thus the bus width to the math units can be increased to accommodate the change. In theory, the fully integrated math units would take no extra time to complete since currently the only time increases are due to the difficulty interfacing with them.

4.1.3 Code Considerations.

To further understand this architecture, the JVM bytecodes used by the actual code were considered. When compiling Java code, the JVM translates the actual Java code of the SubBytes function, seen in Figure 32, to assemble, which can be seen in Figure 33.



```
Native.wr(0x01, Const.IO_LED5);

for(int round = 1; round < 10; round++){

this.state = this.subBytes(this.state);
    for (int j = 0; j < 4; j++)
        for(int dog = 0; dog < 4; dog++)
            state[dog][j] = (byte)S_Box[(state[dog][j]>>4)&0xf][state[dog][j]&0xf];
```

Figure 32. Sub bytes in java code

Looking at the assembler code, the red box labeled "A" represents the translation of the native.wr function (the trigger pin). Red box B is the first for loop which spans the whole encryption algorithm. Red box C is the first of the two "*for*" loops and red box D is the second "for" loop to cycle through all 16 bytes, and red boxes E and F represent the conditional branch statements at the end of the for loops. The code in between the red boxes is the execution of the Java command "$state[dog][j] = (byte)S\_Box[(state[dog][j] >> 4)\&0xf][state[dog][j]\&0xf];$." Of particular note are the stack operations on lines 783 through 824 and the operation on 826 which sends the finished information back to the heap in external RAM. After the code is assembled, it is then compiled to machine language.

4.2 AES Tests

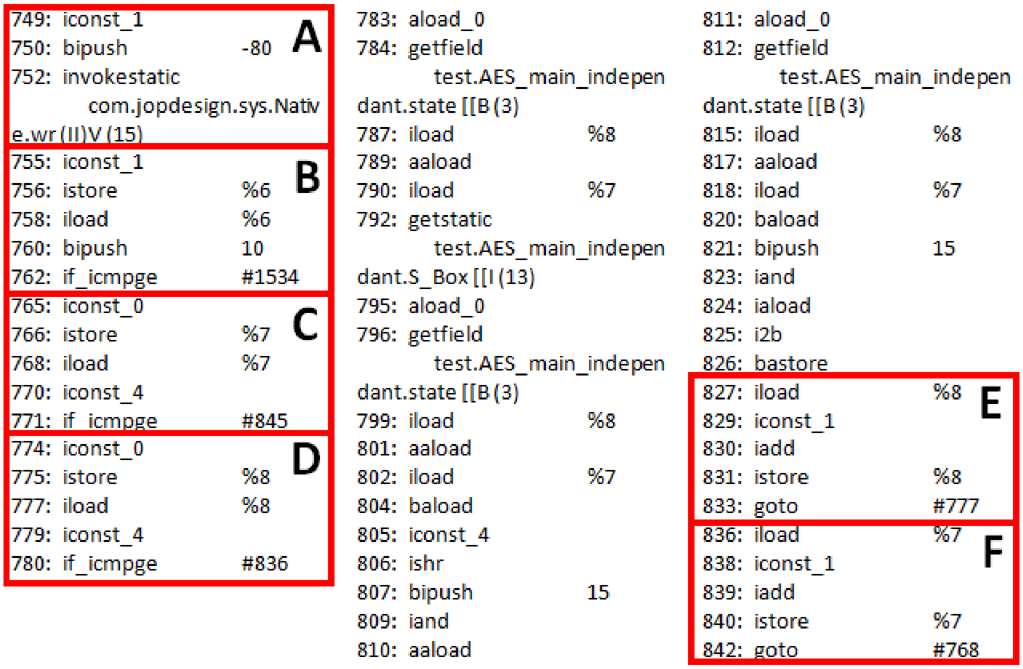
The AES tests were run on the JOP first without any countermeasures, then they were run again with the double stack, and then once more for the double RAM. The clean JOP without countermeasures functioned as the baseline with which to compare the results from the two countermeasures. When conducting all three tests, mapping out the XY spectral plot for the area of analysis (the subbytes function) was conducted first, then the location with the most distinguishable traces was selected. This was done three times for each test resulting in three locations for each test. Although the locations were very close to one another, all located over capacitor C1, they had minor variations in position. At each position 10,000 traces were recorded. From these traces, using every thousand as a different data point, the necessary traces to evaluate the correct key byte for key byte 1 was evaluated. For example, if starting with the first trace, trace 0, if it took 331 traces to get a correct key guess. Then the next data point would start with trace 1000 and see how many traces were needed to get the key guess, and so on and so forth starting at trace 2000, 3000, 4000 and all the way to trace 9000. For all traces, the sampling rate was 1GS/s with a 200Mhz filter. Once the data was collected, it was resampled to a frequency of 99Mhz. Resampling means that if there were previously 10 samples for each tick of the clock, then Inspector would average all 10 samples and replace the 10 samples with one sample representing that clock cycle. This result was then rectified by passing it through an “abs” filter which takes the absolute value of all the samples to make all numbers positive (and increase correlation between samples). Lastly, as can be seen from the traces, when the math calculations are being performed on the data (which used the stack), it is seen as the valleys (see Figure 34), and when values are written to the RAM, it appears as the mountains in the traces (see Figure 35).

4.2.1 JOP Clean.

In testing the JOP without countermeasures, the JOP before modification can be seen in Figure 36 and the trigger associated for when Subbytes is being performed can be seen in 37. Lastly, after being resampled and rectified, the actual trace being considered for security is seen in Figure 38.

The test was repeated three times in three positions over the power capacitor C1. Although the positions were all in the same general area, they were all in slightly different locations. The data points for each position were split up between the portion of execution and the portion of RAM writeback. The results for the minimum number of traces needed for the three positions over the power capacitor during the execution portion of the power trace for the first key byte are seen in Table 8 and the corresponding box plot of the data showing outliers can be seen in Figure 39. Likewise, the results for the minimum number of traces needed for the three positions over the power capacitor during the RAM writeback of the power trace for the first key byte are seen in Table 9 and the corresponding box plot of the data showing outliers can be seen in Figure 40.

An interesting thing to note at this point when only considering the original JOP, the overall average number of traces needed to derive the key during the RAM



```
749:  iconst_1
750:  bipush                -80
752:  invokestatic
           com.jopdesign.sys.Nativ
e.wr (II)V (15)
755:  iconst_1
756:  istore                %6
758:  iload                 %6
760:  bipush                10
762:  if_icmpge             #1534
765:  iconst_0
766:  istore                %7
768:  iload                 %7
770:  iconst_4
771:  if_icmpge             #845
774:  iconst_0
775:  istore                %8
777:  iload                 %8
779:  iconst_4
780:  if_icmpge             #836
```

```
783:  aload_0
784:  getfield
         test.AES_main_indepen
dant.state [[B (3)
787:  iload                %8
789:  aaload
790:  iload                %7
792:  getstatic
         test.AES_main_indepen
dant.S_Box [[I (13)
795:  aload_0
796:  getfield
         test.AES_main_indepen
dant.state [[B (3)
799:  iload                %8
801:  aaload
802:  iload                %7
804:  baload
805:  iconst_4
806:  ishr
807:  bipush               15
809:  iand
810:  aaload
```

```
811:  aload_0
812:  getfield
         test.AES_main_indepen
dant.state [[B (3)
815:  iload                %8
817:  aaload
818:  iload                %7
820:  baload
821:  bipush               15
823:  iand
824:  iaload
825:  i2b
826:  bastore
827:  iload                %8
829:  iconst_1
830:  iadd
831:  istore               %8
833:  goto                 #777
836:  iload                %7
838:  iconst_1
839:  iadd
840:  istore               %7
842:  goto                 #768
```

Figure 33. Subbytes Assembled Code

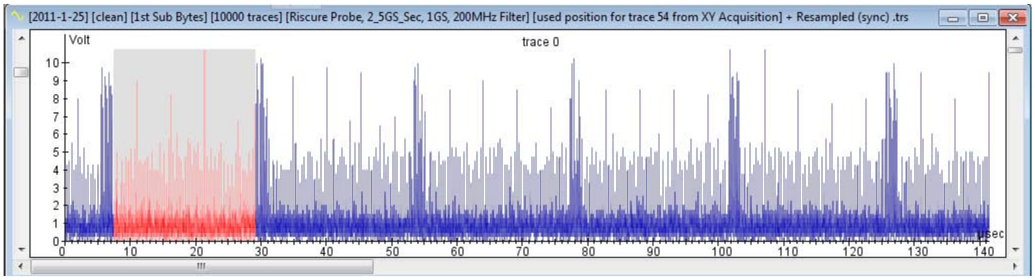

[2011-1-25] [clean] [1st Sub Bytes] [10000 traces] [Riscure Probe, 2_5GS_Sec, 1GS, 200MHz Filter] [used position for trace 54 from XY Acquisition] + Resampled (sync) .trs
Volt
trace 0
10
9
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
µsec

Figure 34. Location of AES attack during execute stage for first key byte

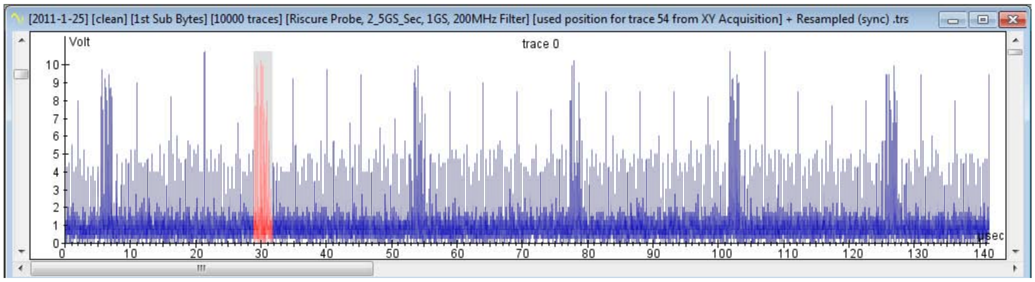

[2011-1-25] [clean] [1st Sub Bytes] [10000 traces] [Riscure Probe, 2_5GS_Sec, 1GS, 200MHz Filter] [used position for trace 54 from XY Acquisition] + Resampled (sync) .trs
Volt
trace 0
10
9
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
µsec

Figure 35. Location of AES attack during RAM writeback for first key byte

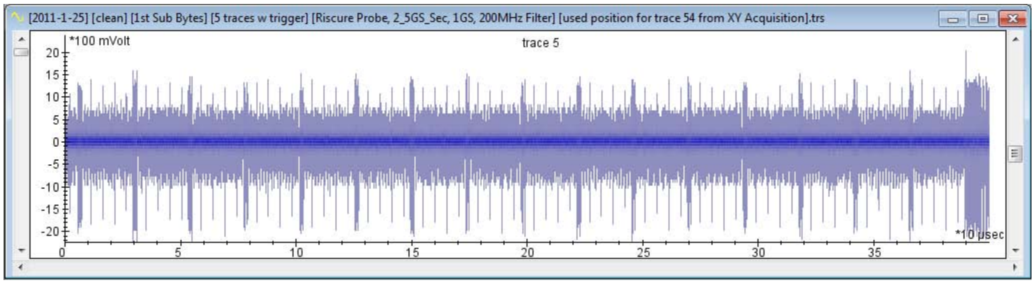


Figure 36. AES trace on unprotected JOP

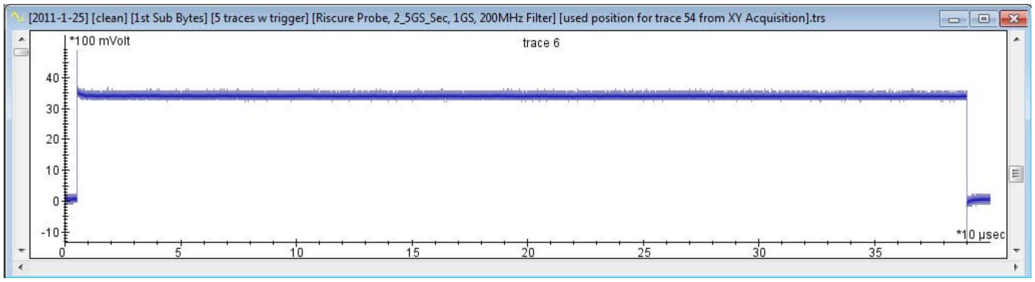


Figure 37. Trigger for AES trace on unprotected JOP

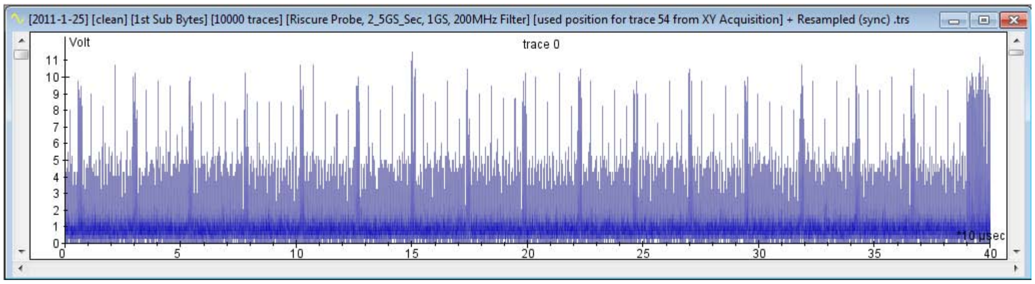


Figure 38. AES trace on unprotected JOP after resampling

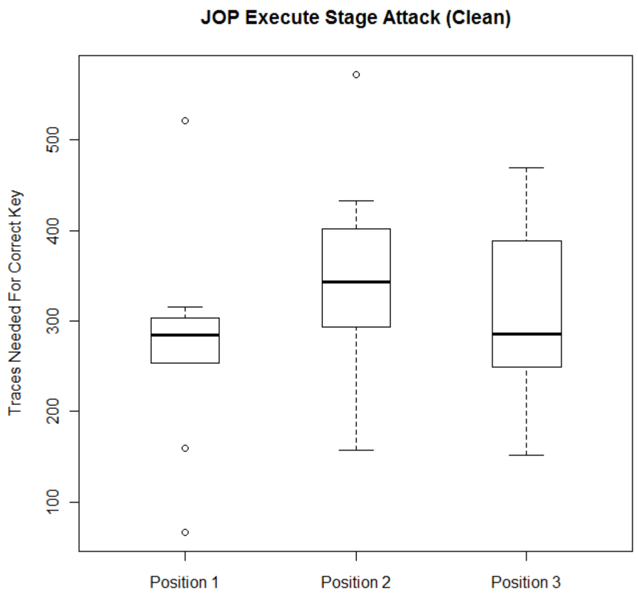


Figure 39. Box Plot of results from unmodified JOP during execution stage

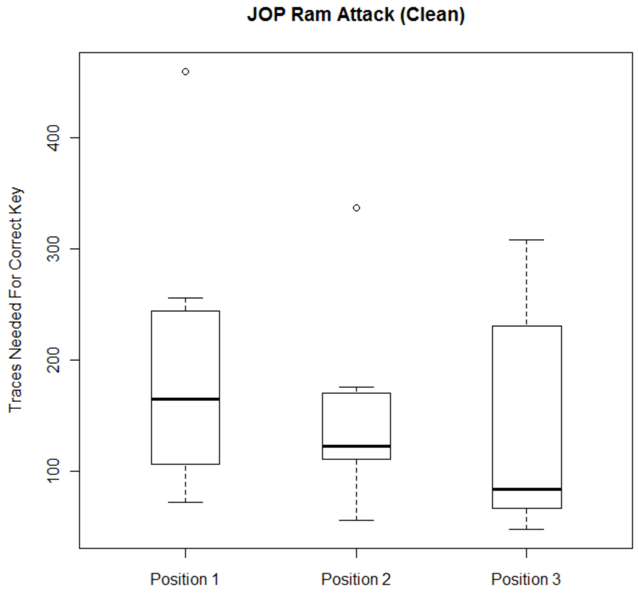


Figure 40. Box Plot of results from unmodified JOP during RAM writeback

Table 8. **Minimum traces needed for correck key using unmodified JOP looking at the execution portion**

	Clean JOP		**Execution portion**
traces	**Position 1 (traces)**	**Position 2 (traces)**	**Position 3 (traces)**
0-999	301	293	269
1000-1999	303	322	249
2000-2999	253	251	389
3000-3999	66	362	311
4000-4999	316	573	192
5000-5999	522	433	303
6000-6999	159	157	251
7000-7999	276	357	470
8000-8999	269	402	152
9000-9999	294	330	425
Average:	275.9	348	301.1
Stdev:	116.505	111.034	101.157
Overall Average:	308.3		

writeback portion of the trace was 154.8 traces and required only about 275 sample of each trace (the highlighted portion in Figure 35), but the overall average number of traces needed to derive the key during the execution portion of the trace was 308.3 traces and needed about 2,050 samples per trace (the highlighted portion of Figure 34). This meant that there was an average of 42 Kilosamples needed during the RAM write back and an average of 632 kilosamples needed during the execution stage. This difference in needed traces is because the values that correlate for the AES SCA attack are only available toward the end of the execution potion and the RAM write back uses a lot more power to interface with the RAM module which is located off the FPGA resulting in strong emissions off the chip.

4.2.2 JOP With Double Stack.

Next, tests were conducted on the JOP using the Double Stack countermeasures. The traces can be seen in Figure 41 and the trigger associated for when Subbytes is

Table 9. Minimum traces needed for correct key using unmodified JOP looking at RAM writeback

	Clean JOP		RAM Writeback
traces	Position 1 (traces)	Position 2 (traces)	Position 3 (traces)
0-999	256	337	308
1000-1999	244	55	230
2000-2999	146	82	94
3000-3999	85	110	72
4000-4999	195	170	60
5000-5999	72	126	70
6000-6999	106	117	116
7000-7999	137	168	232
8000-8999	460	175	66
9000-9999	183	119	47
Average:	188.4	145.9	129.5
Stdev:	114.003	77.477	92.151
Overall Average:	154.6		

being performed can be seen in Figure 42. Lastly, after being resampled and rectified, the actual trace being considered for security is seen in Figure 43.

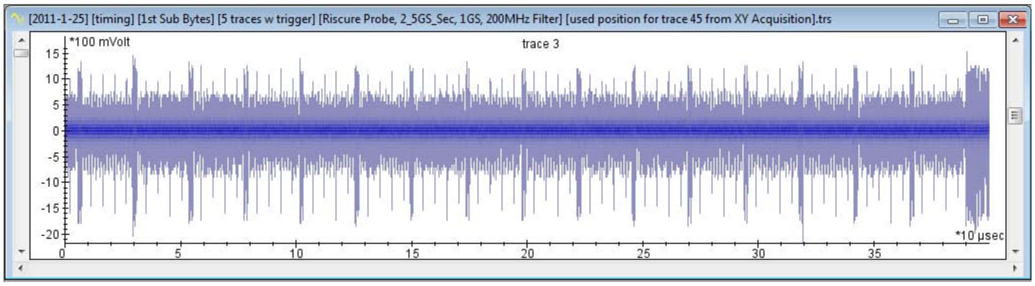

[2011-1-25] [timing] [1st Sub Bytes] [5 traces w trigger] [Riscure Probe, 2_5GS_Sec, 1GS, 200MHz Filter] [used position for trace 45 from XY Acquisition].trs
*100 mVolt
trace 3
15
10
5
0
-5
-10
-15
-20
0
5
10
15
20
25
30
35
*10 µsec

Figure 41. AES trace of JOP with double Stack

The test was repeated three times in three positions over the power capacitor C1. Although the positions were all in the same general area, they were all in slightly different locations. The data points represent only the execution stage of the trace because that is the only portion of the trace that would be effected by the Double Stack. The results for the minimum number of traces needed for the three positions over the power capacitor during the execution portion of the power trace for the first

Table 10. Minimum traces needed for correck key using Double Stack

	Double Stack		Execution portion
traces	Position 1 (traces)	Position 2 (traces)	Position 3 (traces)
0-999	253	285	153
1000-1999	385	276	129
2000-2999	287	207	522
3000-3999	174	227	167
4000-4999	289	404	148
5000-5999	373	263	33
6000-6999	314	147	418
7000-7999	178	330	204
8000-8999	400	246	502
9000-9999	166	69	323
Average:	281.9	245.4	259.9
Stdev:	88.624	92.938	170.166
Overall Average:	262.4		

key byte are seen in Table 10 and the corresponding box plot of the data showing outliers can be seen in Figure 44.

4.2.3 JOP With Double RAM.

Next, tests were conducted on the JOP using the Double RAM countermeasures. The traces can be seen in Figure 45 and the trigger associated for when Subbytes is being performed can be seen in Figure 46. Lastly, after being resampled and rectified, the actual trace being considered for security is seen in Figure 47.

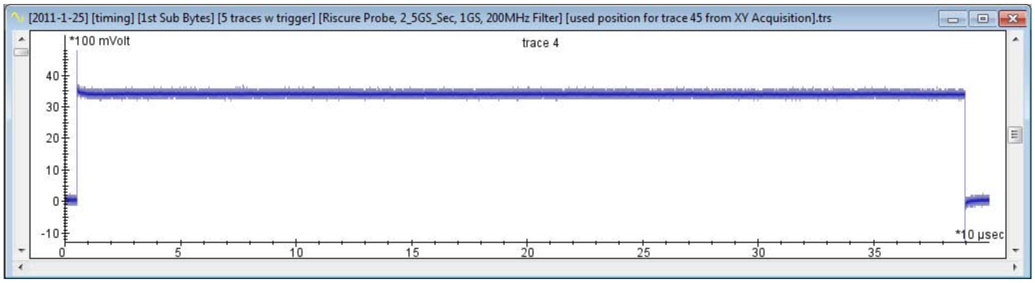
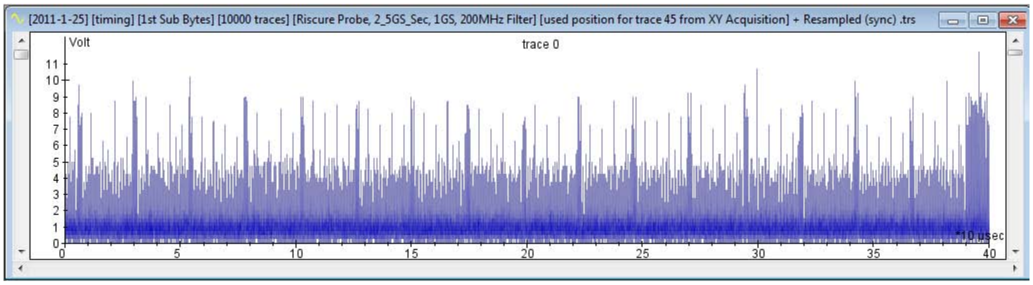

[2011-1-25] [timing] [1st Sub Bytes] [5 traces w trigger] [Riscure Probe, 2_5GS_Sec, 1GS, 200MHz Filter] [used position for trace 45 from XY Acquisition].trs
*100 mVolt
trace 4
40
30
20
10
0
-10
0
5
10
15
20
25
30
35
*10 µsec

Figure 42. Trigger for AES trace of JOP with double Stack



[2011-1-25] [timing] [1st Sub Bytes] [10000 traces] [Riscure Probe, 2_5GS_Sec, 1GS, 200MHz Filter] [used position for trace 45 from XY Acquisition] + Resampled (sync) .trs

Figure 43. AES trace of JOP with double Stack after resampling

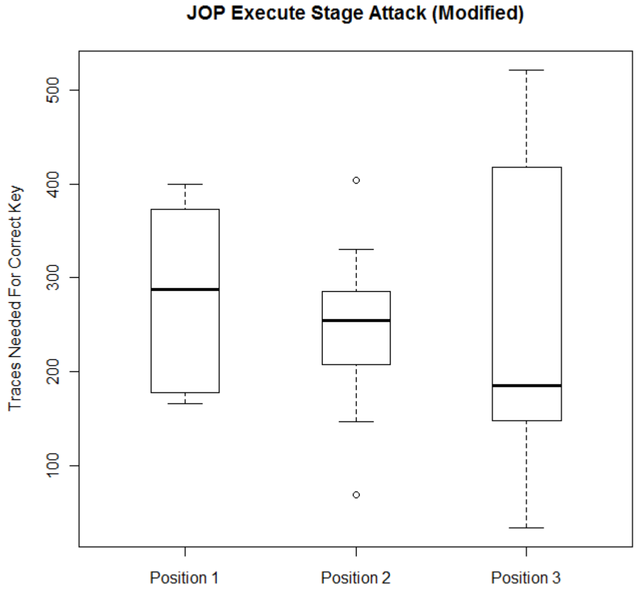
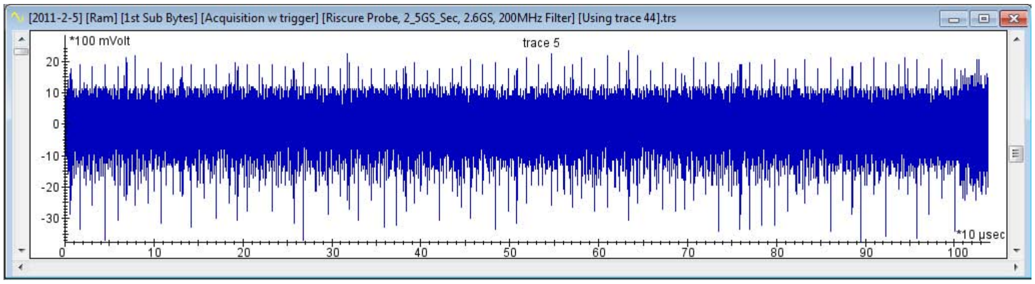


Figure 44. Box Plot of results from JOP with Double Stack during execution stage



[2011-2-5] [Ram] [1st Sub Bytes] [Acquisition w trigger] [Riscure Probe, 2_5GS_Sec, 2.6GS, 200MHz Filter] [Using trace 44].trs

Figure 45. AES trace on JOP with double RAM

Table 11. Minimum traces needed for correck key using JOP with Double RAM

	Double RAM		RAM Writeback
traces	Position 1 (traces)	Position 2 (traces)	Position 3 (traces)
0-999	336	115	161
1000-1999	195	177	312
2000-2999	249	133	175
3000-3999	289	148	231
4000-4999	234	191	133
5000-5999	157	203	281
6000-6999	198	124	314
7000-7999	345	220	355
8000-8999	205	122	238
9000-9999	366	180	162
Average:	257.4	161.3	236.2
Stdev:	72.656	37.594	77.104
Overall Average:	218.3		

The test was repeated three times in three positions over the power capacitor C1. Although the positions were all in the same general area, they were all in slightly different locations. The data points represent only the RAM writeback portion of the trace because that is the only portion of the trace that would be effected by the Double RAM. The results for the minimum number of traces needed for the three positions over the power capacitor during the RAM writeback portion of the power trace for the first key byte are seen in Table 11 and the corresponding box plot of the data showing outliers can be seen in Figure 48.

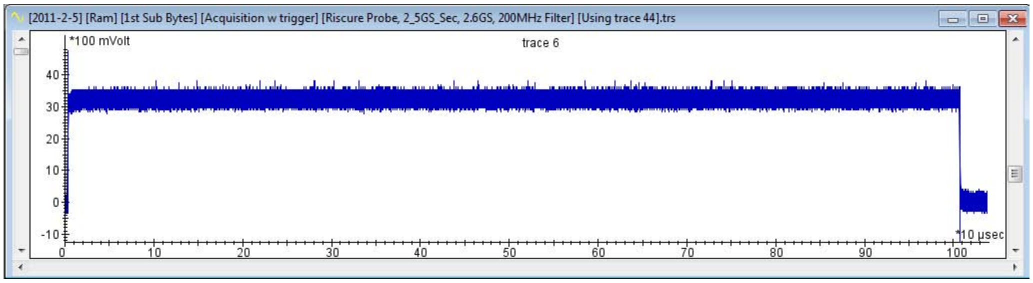

[2011-2-5] [Ram] [1st Sub Bytes] [Acquisition w trigger] [Riscure Probe, 2_5GS_Sec, 2.6GS, 200MHz Filter] [Using trace 44].trs
*100 mVolt
trace 6
40
30
20
10
0
-10
0
10
20
30
40
50
60
70
80
90
100
*10 µsec

Figure 46. Trigger for AES trace on JOP with double RAM

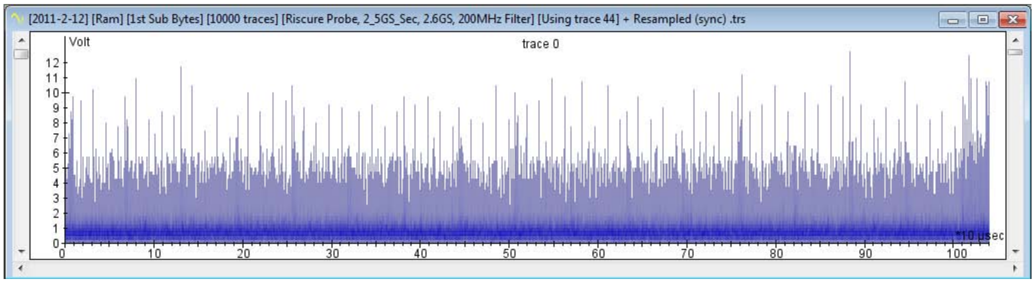

[2011-2-12] [Ram] [1st Sub Bytes] [10000 traces] [Riscure Probe, 2_5GS_Sec, 2.6GS, 200MHz Filter] [Using trace 44] + Resampled (sync) .trs
Volt
trace 0
12
11
10
9
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
60
70
80
90
100
*10 µsec

Figure 47. AES trace on JOP with double RAM after resampling

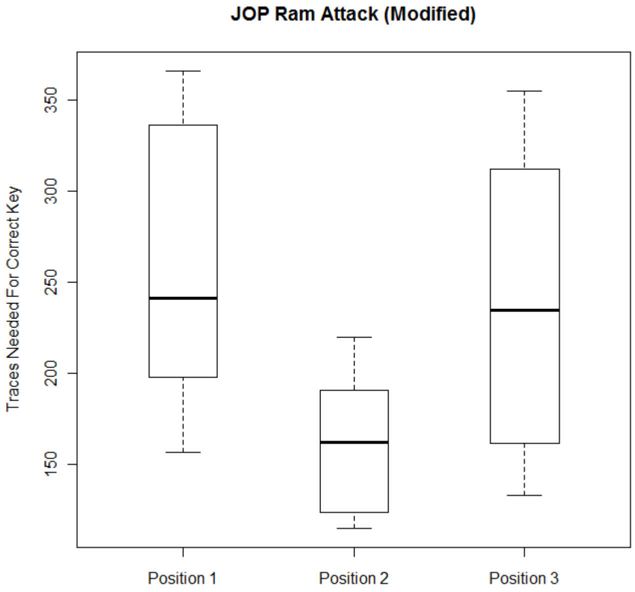


Figure 48. Box Plot of results from JOP with Double RAM during RAM writeback

4.2.4 Obfuscation of Execute Portion of Trace.

Performing a two sided t-test on the results from the unmodified JOP during the execute stage and JOP with the Double Stack during the execute stage after removing the outliers as noted in the box plots above yielded a 95% confidence interval of -95 traces to 14 traces. This means that with a 95% confidence, the difference in the average number of traces for the population will lie somewhere between -95 traces and 14 traces. Put more specifically, because 0 lies in the 95% confidence interval, the results are 95% confident that these data sets are not different. That is to say that the Double Stack modification did not have an appreciable change in the number of required traces for the discovery of the first key byte during the execution portion. This, however, does not come as a very big surprise. The JOP contains many registers that pass the value of the result from the Sbox substitution, and each time the value is passed from one register to the next information is leaked. The goal of the double stack to increase protection was based on the stack having a stronger leakage signal than the registers, and thus having an appreciable change in the security when modified. As it turned out, the information off the stack is not significant when compared to the leakage of the registers in Stack.vhd that hold the value before actually passing it to the stack. Additionally there are the registers in Core.vhd, JOPcpu.vhd, mem_sc.vhd, sdpram.vhd and to JOP_ml50x.vhd which all contain registers that hold the finished value. The only way to fix this problem is to change the underlying structure of the JOP from a Von Newmann architecture to a Harvard architecture. Currently the JOP stores the values for the data and the instructions in the same RAM, but changing the architecture to a Harvard Architecture would mean changing the JOP such that instructions and data are saved in two separate memory spaces. Allowing the data to stay split into two corresponding variables right up until execution on the data, and then re-split into two values again after execution and be saved that way

Table 12. Results from a two sided T-test on the execution stage of subbytes when comparing the unmodified JOP to the JOP with the double stack without outliers shown in Box Plots

t-value	-1.5061
degrees of freedom	52
p-value	0.1381
95 percent confidence interval:	-95.043 13.543
sample mean estimates:	264.25 305.00

in the main data memory would allow all registers in the JOP to be obfuscated. Due to the JOP having a Von Newmann architecture and thus an inability to distinguish instructions from data in the main memory, an instruction cannot be split between two values until the execute phase because it would cause instructions to be decoded wrongly and the JOP to crash.

4.2.5 Obfuscation of the RAM write back of the Trace.

Performing a two sided t-test on the results from the unmodified JOP during the RAM writeback and the Double RAM during the RAM writeback after removing the outliers as noted in the box plots above yielded a 95% confidence interval of 43 traces to 137 traces (see Table 13). This means that with a 95% confidence, the difference in the average number of traces for the population will lie somewhere between 43 traces and 137 traces. Put more specifically, the Double RAM had a substantial improvement in the security for the RAM writeback portion of the trace. This gives us an increase in the number of traces over the unmodified JOP to be between 31% increased security and 87% increased security with a 95% confidence. This also comes as no surprise. As mentioned in the previous section about the JOP without countermeasures, there was a significantly greater leakage of information during the RAM write back, requiring 7% of the samples needed during the execution stage (42KS vs

Table 13. Results from a two sided T-test on the execution stage of subbytes when comparing the unmodified JOP to the JOP with the double stack without outliers shown in Box Plots

t-value	4.2493	
degrees of freedom	56	
p-value	8.187e-05	
95 percent confidence interval:	42.878	137.178
sample mean estimates:	264.25	305.00

632KS). With this knowledge, we know that the RAM write back module leaks significantly more information than the regular registers inside the JOP. Knowing this, obfuscating the leakage off the RAM write back module has a measureable increase in security.

4.3 RSA Tests

4.3.1 JOP without countermeasures.

As mentioned in Chapter 2, Java uses bit group exponentiation to accomplish the RSA module. To attack this algorithm, the traces need to be prepared such that only the multiplies are evident and are large enough such that given a threshold, Inspector can "resample" each multiply operation and replace it with a single data point.

In collecting data, the oscilloscope was set to trigger during just the exponentiation, and the traces acquired can be seen in Figure 49 using the trigger seen in Figure 50. Like earlier, this sample group was originally recorded at 1GS/s, and then rectified and resampled, which the resulting trace can be seen in Figure 51. It was seen that spikes in the trace occurred at a 3Mhz sample frequency, so this trace was resampled again to an even lower frequency to prepare it for a weighted low pass filter, and the resulting trace can be seen in Figure 52. Then this graph was filtered (see Figure 53) and then resampled in such a way that each multiply was recorded as one trace for analysis, resulting in Figure 54.

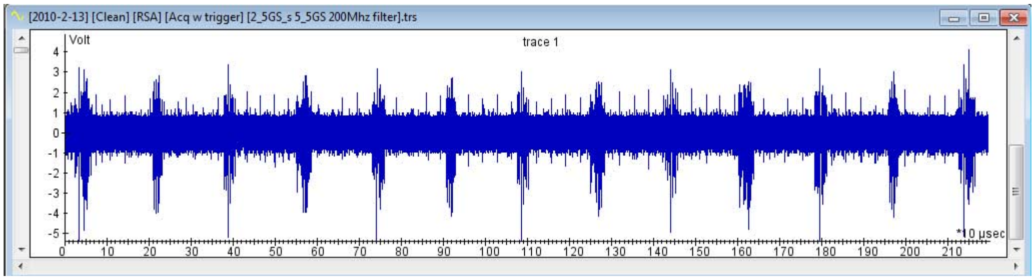


Figure 49. Plot of RSA bitgroup exponentiation without countermeasures

At this point, Inspector was used to correlate and find the binary key. The actual key was 0x1D however Inspector found a key of 0x1B. Upon further investigation, it was found through looking at Inspector files on the RSA high order analysis that Inspector employs bitgroup exponentiation differently. As earlier explained in Chapter 2, Java first calculates all the odd powers of the base number and optimizes out the

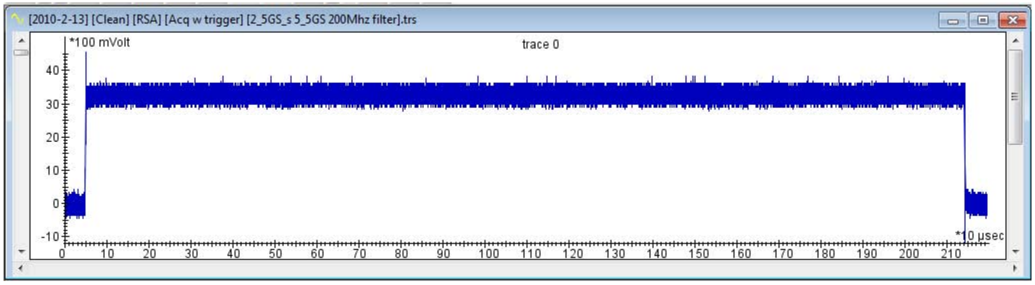


Figure 50. Plot of trigger for RSA bitgroup exponentiation without countermeasures

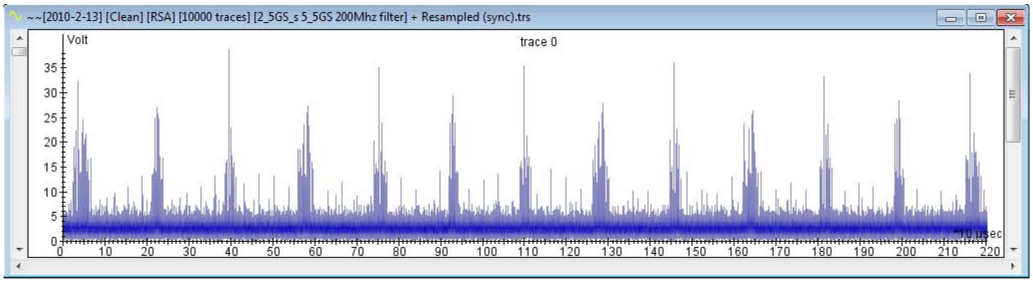


Figure 51. Plot of RSA bitgroup exponentiation without countermeasures after resampling

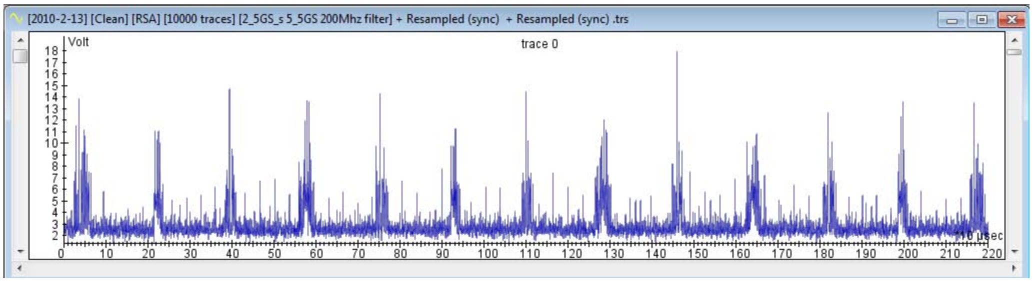


Figure 52. Plot of RSA bitgroup exponentiation without countermeasures after second resampling

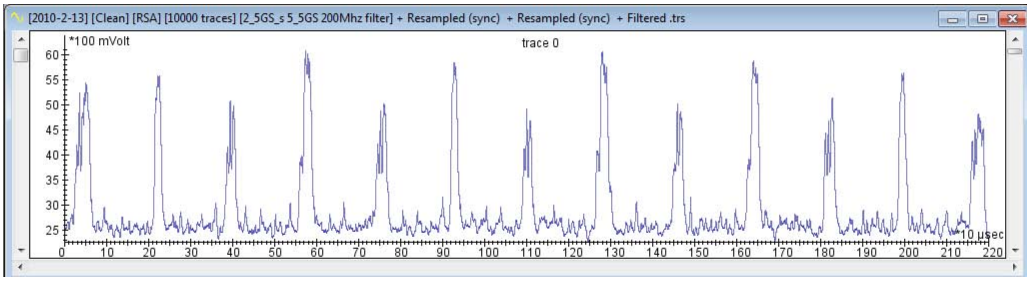


Figure 53. Plot of RSA bitgroup exponentiation without countermeasures after a lowpass filter

even powers of the base number. The optimization occurs by knowing that multiplying a number by a second number, then squaring the product ($a = (b * c)^2$) is the same as saying squaring the first number, and multiplying that by the square of the second number ($a = b^2 * c^2$). Thus, when java encounters the need to multiply by an even exponent, say c^6, then it first multiplies by c^3 and then squares the result. Inspector does not use this optimization and expects both even and odd powers to be used in exponentiation. For example, if the exponent bit sequence "110" is encountered, java will perform a square, square, multiply by c^3 and then square it the third time. On the other hand, Inspector is expected that when a "110" is encountered, the cryptosystem will perform a square square square and then multiply by c^6. It gives the same result in the end, except that Inspector is not set up to derive a key for a bitgroup exponentiation that has been optimized. In an attempt to get Inspector to function with the optimized version, a key was used that contained only odd powers (0x39). This key causes Java to perform three squares, then multiply by n^7, then perform 3 more squares and multiply by n. The resulting trace can be seen in 55. Unfortunately, it was brought to light that another problem existed in using the Inspector module - peaks are generated for the squares and the multiplies like expected by Inspector, but then they are also generated for the Modulus function which calls an algorithm that is similar to the division algorithm. Because Inspector is not set up to handle this situation either, the ability for Inspector to be able to

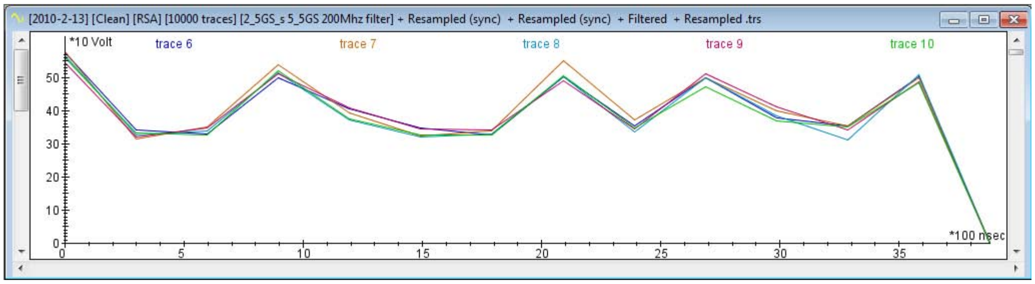


Figure 54. Plot of RSA bitgroup exponentiation without countermeasures after the filtered resampling

generate a correct key was dropped in favor of looking at the filtered resampled trace for an overt pattern. As can be seen in Figures 54 and 55 there is an overt pattern that all traces are following. Thus this version of RSA is not protected.

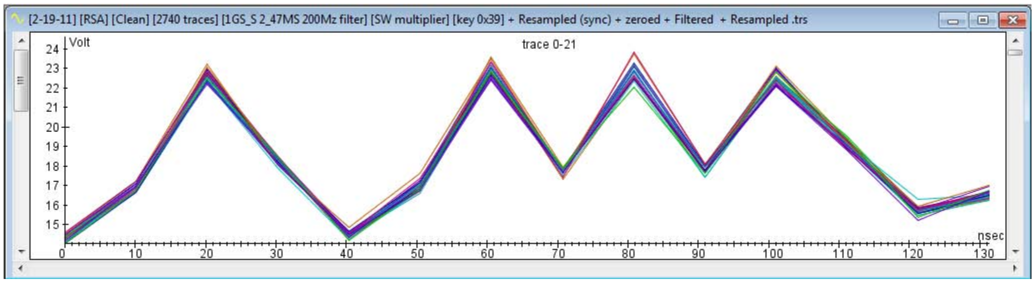

[2-19-11] [RSA] [Clean] [2740 traces] [1GS_S 2_47MS 200Mz filter] [SW multiplier] [key 0x39] + Resampled (sync) + zeroed + Filtered + Resampled .trs
Volt
trace 0-21
24
23
22
21
20
19
18
17
16
15
nsec
0
10
20
30
40
50
60
70
80
90
100
110
120
130

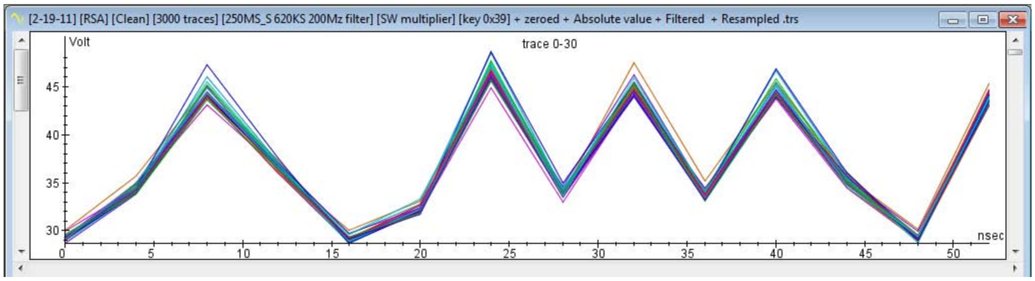
Figure 55. Plot of RSA bitgroup exponentiation without countermeasures after the filtered resampling

The last test run on the unmodified version of the JOP was to see if it was possible to see the same information if the sample rate on the oscilloscope was reduced to 250MS/s and everything else was kept the same. The resulting trace can be seen in Figure 56. This was in important test because the version of the JOP with the DAIL-MOM takes significantly longer due to having to write the values to ports and thus the fastest it could be sampled was 250MS/s. Although in the previous spectrums, frequencies higher than 125 Mhz did exist in small number, those frequencies would be aliased at this slower sampling frequency effectively creating more noise. When the same test was run at the slower frequency, effectively the same results were found (as compared to the same data at the higher frequency as seen in Figure 55), showing that the results at 250MS/s are still valid.

4.3.2 JOP with Double Adder.

Next, the JOP was analyzed with the Double Adder, and the traces acquired can be seen in Figure 57 using the trigger seen in Figure 58. Like earlier, this sample group was originally recorded at 1GS/s, and then was rectified and resampled, which the resulting trace can be seen in Figure 59. At this point, there is a clear increase in noise due to the Double Adder. This significant increase in the noise renders these trace unable to use the previous method of analysis because an acceptable threshold above the noise could not be reached such that a resampling of the multiplies could not happen. After several attempts at different options, it was found that by manually zeroing out the portions in between the multiplication spikes that contained noise, resulted in Figure 60 and then running the filter resulted in the trace seen in Figure 61 the final resampling could be used resulting in the Figure 62. Unfortunately, as seen previously with the unprotected JOP, using the Double adder along with the synchronous hardware multiplier that came with the JOP did not obfuscate the traces.

As a result, it was thought that perhaps by using a simple software multiplier some measure of obfuscation would result because the multiplies would then rely on the Double Adder when calculating a value. Figure 63 shows the code for the software multiplier where first the sign of the result is calculated, and then the code serially goes through all 32 bits adding and shifting when necessary. This change resulted in the resampled trace seen in Figure 64 and the filtered resampled trace seen in Figure 65. Unfortunately, there was still no obfuscation even though the multiply unit relied on the double adder for calculations. After running these tests, it became obvious why there was no difference in the traces though. The double adder alternates every time in a round robin fashion, and each multiply operation calls 32 adds, every multiply



[2-19-11] [RSA] [Clean] [3000 traces] [250MS_S 620KS 200Mz filter] [SW multiplier] [key 0x39] + zeroed + Absolute value + Filtered + Resampled .trs

Figure 56. Plot of RSA bitgroup exponentiation without countermeasures after the filtered resampling on a sample with an original oscilloscope sampling rate of 250Mhz

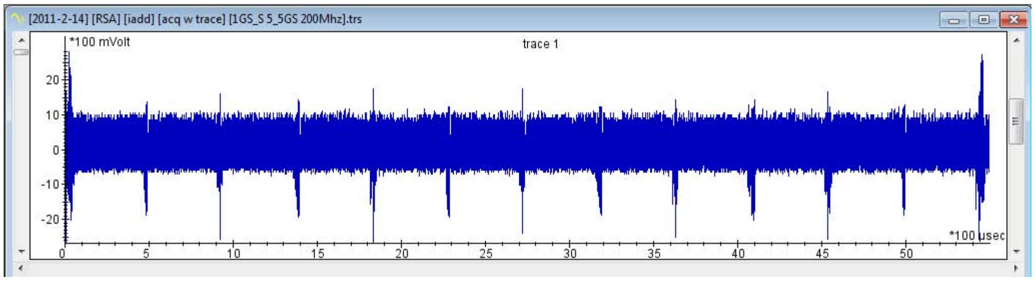


Figure 57. Plot of RSA bitgroup exponentiation with Double Adder

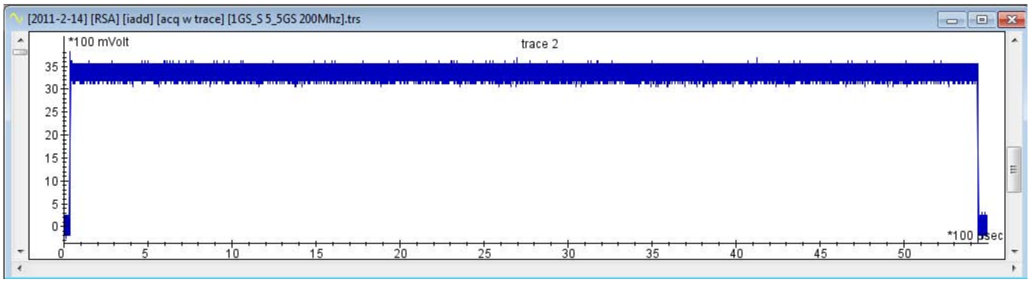


Figure 58. Plot of trigger for RSA bitgroup exponentiation with Double Adder

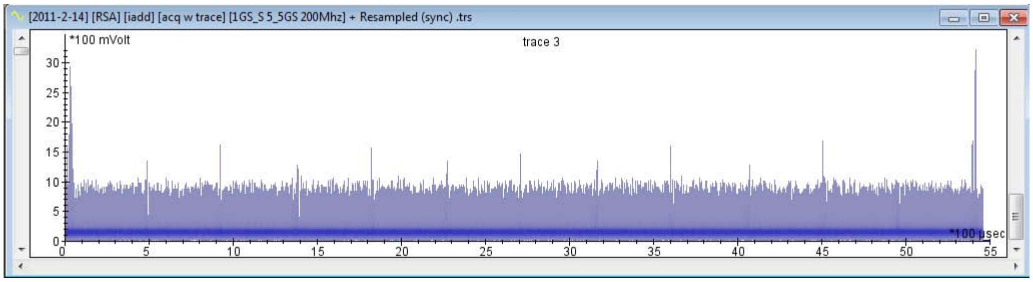


Figure 59. Plot of RSA bitgroup exponentiation with Double Adder after resampling

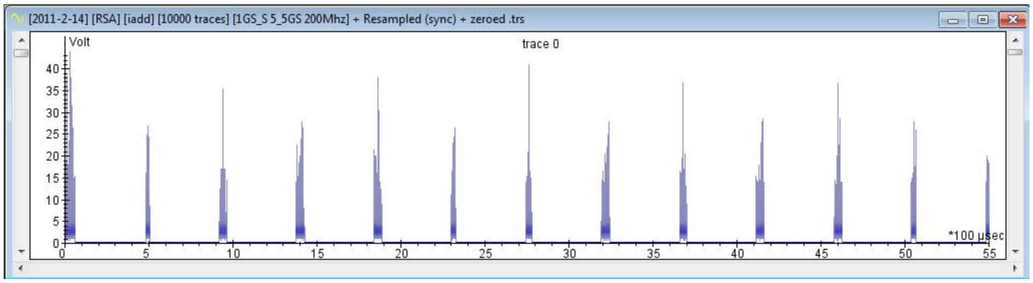


Figure 60. Plot of RSA bitgroup exponentiation with Double Adder after zeroing out noise by hand

operation will have 16 adds with the Carry Look Ahead adder and 16 adds with the Ripple Carry Adder. Thus, the average trace for each multiply would be the same, and there would not be any noticeable obfuscation.

4.3.3 JOP with DAILMOM.

The last test conducted for the RSA countermeasures was to implement the JOP with the DAILMOM. The test resulted with the initial trace (Figure 66) with the corresponding trigger (Figure 67). This was then resampled as previously done with the JOP without countermeasures resulting in the trace in Figure 68. Because the noise was low in-between multiplies like in the original JOP, it was possible to resample it again to reduce the frequency further and the resulted in the trace in Figure 69. This was then filtered resulting in the trace shown in Figure 70. This filtered trace was then resampled once more resulting in the final trace shown in Figure 71.

The final result, shown in Figure 71 shows that the DAILMOM had a significant effect on the multiplications. Where previously in the clean JOP (Figures 54, 55, and 56), and the JOP with the Dual Adder (Figures 62 and 65) there was a noticeable pattern that multiplies and squares followed, when using the DAILMOM this pattern is obfuscated showing success of the DAILMOM in obfuscating the bitgroup exponentiation of RSA.

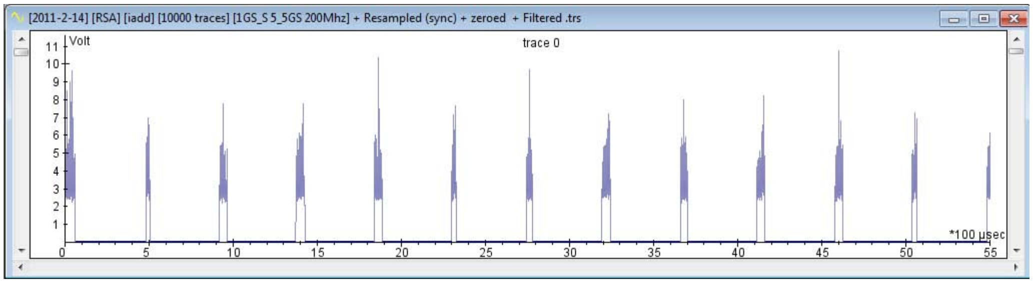


Figure 61. Plot of RSA bitgroup exponentiation with Double Adder after a lowpass filter

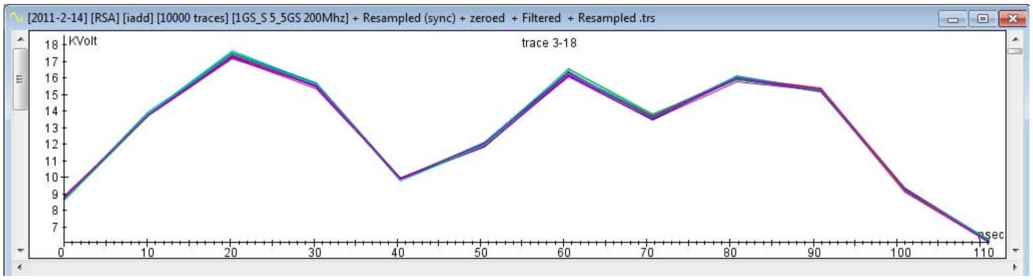
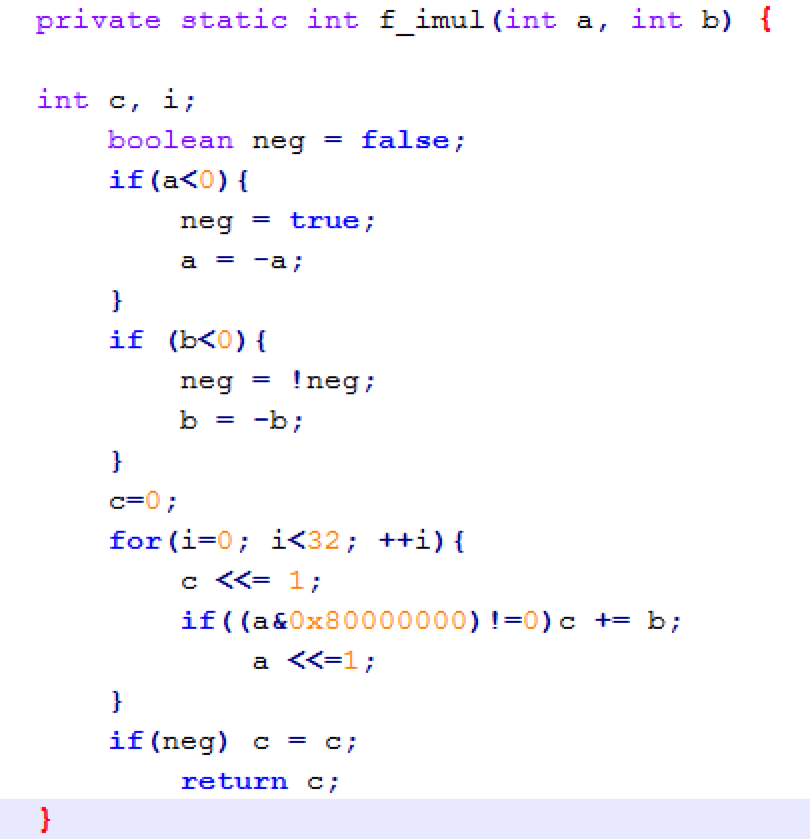


Figure 62. Plot of RSA bitgroup exponentiation with Double Adder after the filtered resampling



```
private static int f_imul(int a, int b) {

int c, i;
     boolean neg = false;
     if(a<0){
          neg = true;
          a = -a;
     }
     if (b<0){
          neg = !neg;
          b = -b;
     }
     c=0;
     for(i=0; i<32; ++i){
          c <<= 1;
          if((a&0x80000000)!=0)c += b;
               a <<=1;
     }
     if(neg) c = c;
          return c;
 }
```

Figure 63. Software version of a multiplier

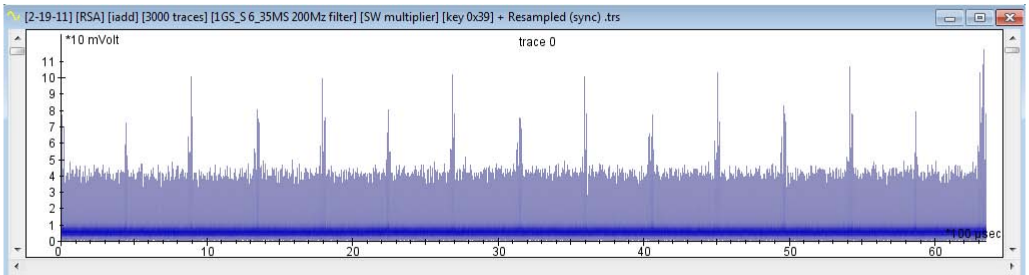


Figure 64. Plot of RSA bitgroup exponentiation with Double Adder using software multiplier after resampling

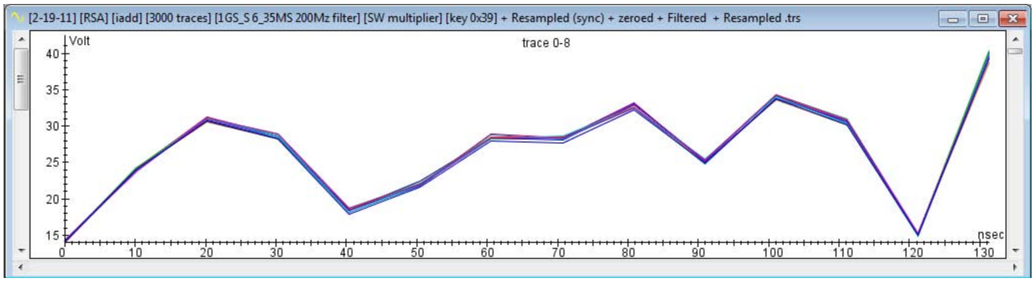


Figure 65. Plot of RSA bitgroup exponentiation with Double Adder using software multiplier after the filtered resampling

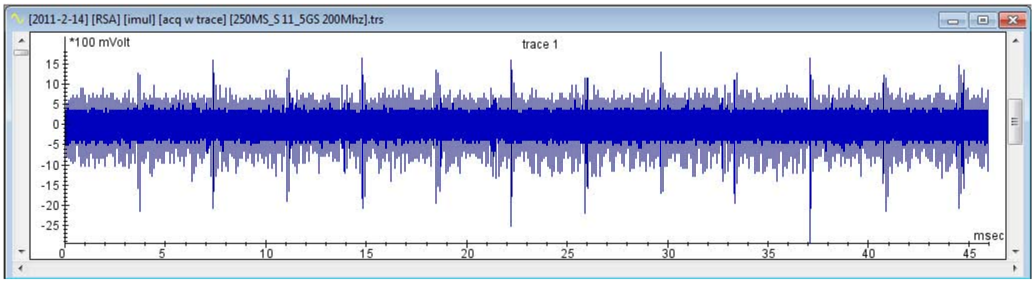


Figure 66. Plot of RSA bitgroup exponentiation with DAILMOM

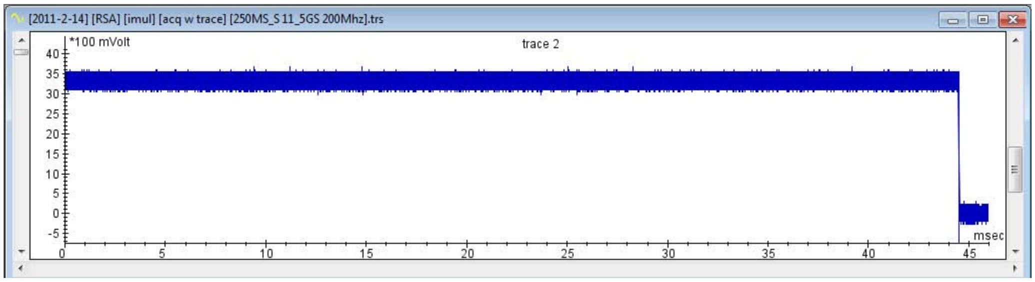


Figure 67. Plot of trigger for RSA bitgroup exponentiation with DAILMOM

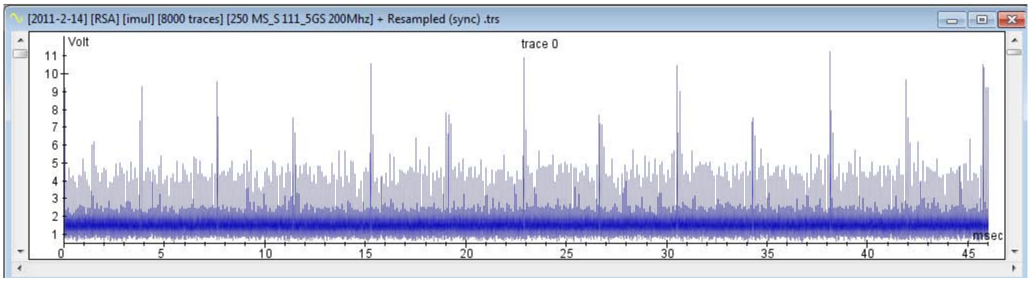


Figure 68. Plot of RSA bitgroup exponentiation with DAILMOM after resampling

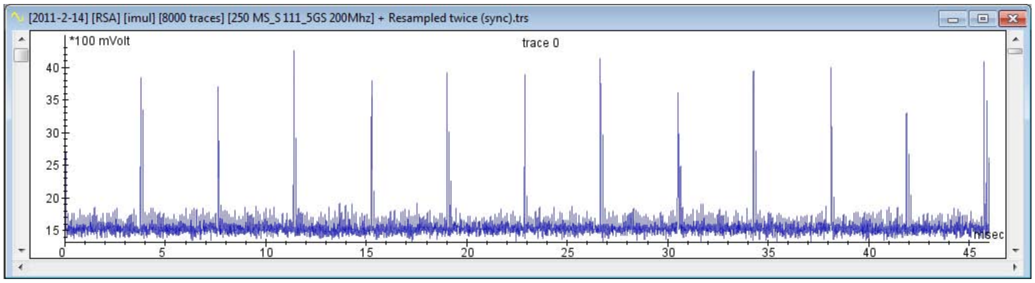


Figure 69. Plot of RSA bitgroup exponentiation with DAILMOM after resampling a second time

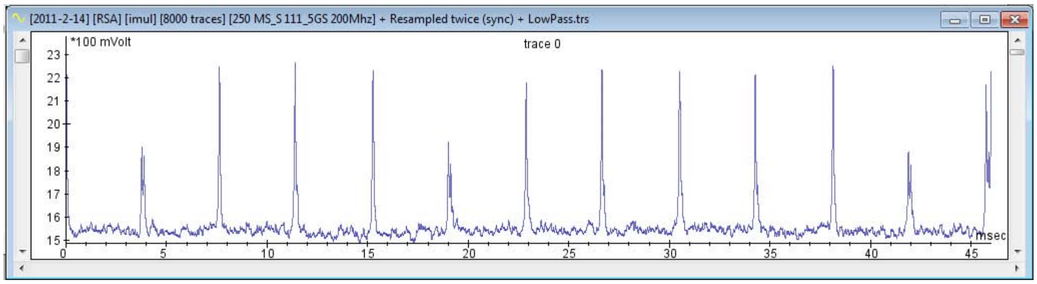


Figure 70. Plot of RSA bitgroup exponentiation with DAILMOM after a lowpass filter

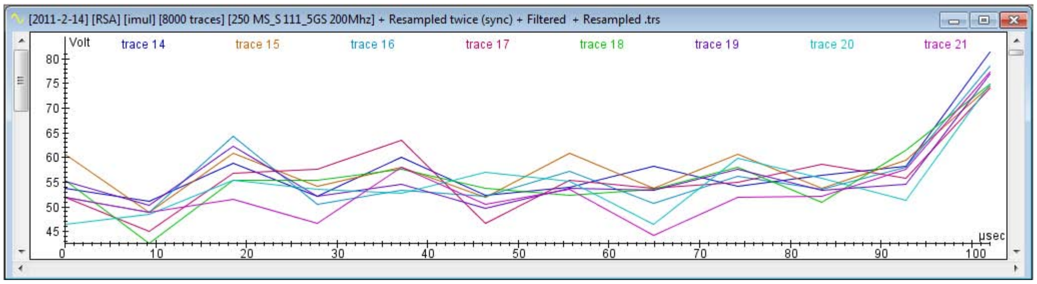


Figure 71. Plot of RSA bitgroup exponentiation with DAILMOM after the filtered resampling

V. Conclusions and Future Work

Chapter 5 is divided into the conclusions of the work completed and the proposals for future work.

5.1 Conclusions

With cryptography being susceptible to SCA attacks, great caution should be taken to ensure that a given cryptography implementation on a given hardware is not at high risk of being compromised. To this end, this research's purpose is to provide a novel and effective way to prevent or deter the compromise an AES or RSA software implementation.

5.1.0.1 AES Conclusions.

For AES encryption it is found that using the double stack did not have a statistical increase in security. The lack of security increase for the double stack was because it protected only one of many different registers on the chip that leak SCA information about the value. Before the change to the stack, it too leaked SCA information just like the other registers on the JOP. After the change to the stack, the stack no longer leaked the SCA information like before, but the other registers still did. Because only the stack was changed, and the other registers still leaked the information as before, the increase in security due to the stack change was negligible. To correct this problem, the underlying data structure of the JOP would need to be changed. Currently the JOP employs a Von Neuman architecture where both the instructions and the data are both saved in the same RAM. If the JOP structure was instead changed to a Harvard Architecture, where the instructions and data are saved in two different locations, it would be possible to split the data values and save them split in the double RAM, and keep them split as they move through the JOP all the way to the execute phase when they would be temporarily “re-assembled” and used for calculations. After the calculation was complete, the value would be re-split and kept in two halves that each had the same hamming weight. The importance of the change is that the registers that still leaked information about the value even after the change in the stack, would not longer leak any information. This is not currently possible in the JOP because when a value is read from the RAM, instructions and data are indistinguishable and obfuscating instructions would cause a runtime error. Changing the underlying architecture could reasonably increase the protection of the JOP several orders of magnitude, making the JOP 100 or 1000 times more secure by removing all the leaky registers on the JOP. On the other hand, the Dual RAM did have a measureable difference in security. With a 95% confidence, it was seen that there was between 31% to 87% increase in the number of traces necessary for Inspector to find the correct key.

5.1.0.2 RSA Conclusions.

Because the RSA results could not depend on Inspector’s ability to find the key from the traces, SPA was used to note the correlation of the data to a recognizable bitgroup pattern. Although implementing the Double Adder did greatly increase the noise of the function, a way was found to zero out the noise and still do an analysis on the results from the Double Adder. As it turned out, regardless of using the external synchronous multiplier, or a software multiplier that employed the Double Adder, the traces still formed a recognizable pattern. The reason for this was simple, in the case of using the external synchronous adder that came with the JOP, no obfuscation occurred during the multiplications because the Dual Adder was not being used during the multiplications. Then, when employing the software multiplication method, the double also had a negligible effect because each adder was both being called 16 times for each multiply, thus the average for each multiply was always the same. On the other hand, after testing the JOP with the DAILMOM, and processing the results, it was found that employing the DAILMOM actually did obviously remove the correlation between the data and the exponent, showing an obvious increase in obfuscation and thus an increase in security.

5.2 Future Work

Future work following this research can take two directions, either further research can be done into finding methods to further obfuscate the JOP or methods can be developed to better increase the security of the dual hardware.

Research to further increase SCA security of the JOP can be done to find new methods to obfuscate the vulnerabilities found in this thesis, or finding new vulnerabilities. For example, one could look into changing the underlying structure of the JOP into a Harvard Architecture and look into options for increase the security as such. Additionally, research can be done to test for the effect on security of the proposed countermeasures while a crypto-processor is running the Data Encryption Standard (DES) cryptography algorithm.

Research to further the four double hardware outlined in this thesis could be done to increase them to have more hardware units, for example to have a triple adder, or split values three ways and employ a triple RAM. These results could be compared and contrasted with the results of having only two hardware units. Furthermore, research can be done to find other hardware units that would benefit from being doubled.

Appendix A. Java Modulo Exponentiation Functions

Listing A.1. ModPow(exponent, modulous)

```
public BigInteger modPow(BigInteger exponent, BigInteger m) {
     if (m.signum <= 0)
         throw new ArithmeticException("BigInteger: modulus not...
             positive");

     // Trivial cases
     if (exponent.signum == 0)
         return (m.equals(ONE) ? ZERO : ONE);

     if (this.equals(ONE))
         return (m.equals(ONE) ? ZERO : ONE);

     if (this.equals(ZERO) && exponent.signum >= 0)
         return ZERO;

     if (this.equals(negConst[1]) && (!exponent.testBit(0)))
         return (m.equals(ONE) ? ZERO : ONE);

     boolean invertResult;
     if ((invertResult = (exponent.signum < 0)))
         exponent = exponent.negate();

     BigInteger base = (this.signum < 0 || this.compareTo(m) >=...
         0
                        ? this.mod(m) : this);
     BigInteger result;


     if (m.testBit(0)) { // odd modulus
         result = base.oddModPow(exponent, m);
     } else {
         /*
          * Even modulus.  Tear it into an "odd part" (m1) and ...
             power of two
          * (m2), exponentiate mod m1, manually exponentiate ...
             mod m2, and
          * use Chinese Remainder Theorem to combine results.
          */

         // Tear m apart into odd part (m1) and power of 2 (m2)
         int p = m.getLowestSetBit();   // Max pow of 2 that ...
            divides m


         BigInteger m1 = m.shiftRight(p);  // m/2**p
         BigInteger m2 = ONE.shiftLeft(p); // 2**p
         // Calculate new base from m1
         BigInteger base2 = (this.signum < 0 || this.compareTo(...
            m1) >= 0
                            ? this.mod(m1) : this);



         // Caculate (base ** exponent) mod m1.
         BigInteger a1 = (m1.equals(ONE) ? ZERO :
                         base2.oddModPow(exponent, m1));

         // Calculate (this ** exponent) mod m2
         BigInteger a2 = base.modPow2(exponent, p);



         // Combine results using Chinese Remainder Theorem
         BigInteger y1 = m2.modInverse(m1);
         BigInteger y2 = m1.modInverse(m2);

         result = a1.multiply(m2).multiply(y1).add
                  (a2.multiply(m1).multiply(y2)).mod(m);
     }

     return (invertResult ? result.modInverse(m) : result);
 }
```

Listing A.2. OddModPow(exponent, modulous)

```
/**
  * Returns a BigInteger whose value is x to the power of y mod...
      z.
  * Assumes: z is odd && x < z.
  */
 private BigInteger oddModPow(BigInteger y, BigInteger z) {
 /*
  * The algorithm is adapted from Colin Plumb's C library.
  *
  * The window algorithm:
  * The idea is to keep a running product of b1 = n^(high-order...
      bits of exp)
  * and then keep appending exponent bits to it.  The following...
      patterns
  * apply to a 3-bit window (k = 3):
  * To append   0: square
  * To append   1: square, multiply by n^1
  * To append  10: square, multiply by n^1, square
  * To append  11: square, square, multiply by n^3
  * To append 100: square, multiply by n^1, square, square
  * To append 101: square, square, square, multiply by n^5
  * To append 110: square, square, multiply by n^3, square
  * To append 111: square, square, square, multiply by n^7
  *
  * Since each pattern involves only one multiply, the longer ...
    the pattern
  * the better, except that a 0 (no multiplies) can be appended...
     directly.
  * We precompute a table of odd powers of n, up to 2^k, and ...
    can then
  * multiply k bits of exponent at a time.  Actually, assuming
  * exponents, there is on average one zero bit between needs ...
    to
  * multiply (1/2 of the time there's none, 1/4 of the time ...
    there's 1,
  * 1/8 of the time, there's 2, 1/32 of the time, there's 3, ...
    etc.), so
  * you have to do one multiply per k+1 bits of exponent.
  *
  * The loop walks down the exponent, squaring the result ...
    buffer as
  * it goes.  There is a wbits+1 bit lookahead buffer, buf, ...
    that is
  * filled with the upcoming exponent bits.  (What is read ...
    after the
  * end of the exponent is unimportant, but it is filled with ...
    zero here.)
  * When the most-significant bit of this buffer becomes set, i...
    .e.
  * (buf & tblmask) != 0, we have to decide what pattern to ...
    multiply
  * by, and when to do it.  We decide, remember to do it in ...
    future
  * after a suitable number of squarings have passed (e.g. a ...
    pattern
  * of "100" in the buffer requires that we multiply by n^1 ...
    immediately;
  * a pattern of "110" calls for multiplying by n^3 after one ...
    more
  * squaring), clear the buffer, and continue.
  *
  * When we start, there is one more optimization: the result ...
    buffer
  * is implcitly one, so squaring it or multiplying by it can ...
    be
  * optimized away.  Further, if we start with a pattern like "...
    100"
  * in the lookahead window, rather than placing n into the ...
    buffer
  * and then starting to square it, we have already computed n...
    ^2
  * to compute the odd-powers table, so we can place that into
  * the buffer and save a squaring.
  *
```

```
    * This means that if you have a k-bit window, to compute n^z,
    * where z is the high k bits of the exponent, 1/2 of the time
    * it requires no squarings.  1/4 of the time, it requires 1
    * squaring, ... 1/2^(k-1) of the time, it reqires k-2 ...
       squarings.
    * And the remaining 1/2^(k-1) of the time, the top k bits are...
        a
    * 1 followed by k-1 0 bits, so it again only requires k-2
    * squarings, not k-1.  The average of these is 1.  Add that
    * to the one squaring we have to do to compute the table,
    * and you'll see that a k-bit window saves k-2 squarings
    * as well as reducing the multiplies.  (It actually doesn't
    * hurt in the case k = 1, either.)
    */
       // Special case for exponent of one


       if (y.equals(ONE))
           return this;

       // Special case for base of zero
       if (signum==0)
           return ZERO;

       int[] base = new int[mag.length];//mag.clone();

               for(int i = 0; i<mag.length; i++)
                       base[i] = mag[i];

       int[] exp = y.mag;
       int[] mod = z.mag;
       int modLen = mod.length;



       // Select an appropriate window size
       int wbits = 0;
       int ebits = bitLength(exp, exp.length);
       // if exponent is 65537 (0x10001), use minimum window size
       if ((ebits != 17) || (exp[0] != 65537)) {
           while (ebits > bnExpModThreshTable[wbits]) {
               wbits++;
           }
       }

       // Calculate appropriate table size
       int tblmask = 1 << wbits;

       // Allocate table for precomputed odd powers of base in ...
          Montgomery form
       int[][] table = new int[tblmask][];
```

```
        for (int i=0; i<tblmask; i++)
            table[i] = new int[modLen];

        // Compute the modular inverse
        int inv = -MutableBigInteger.inverseMod32(mod[modLen-1]);

        // Convert base to Montgomery form
        int[] a = leftShift(base, base.length, modLen << 5);

        MutableBigInteger q = new MutableBigInteger(),
                          a2 = new MutableBigInteger(a),
                          b2 = new MutableBigInteger(mod);

        MutableBigInteger r = a2.divide(b2, q);
        table[0] = r.toIntArray();

        // Pad table[0] with leading zeros so its length is at ...
           least modLen
        if (table[0].length < modLen) {
           int offset = modLen - table[0].length;
           int[] t2 = new int[modLen];
           for (int i=0; i<table[0].length; i++)
               t2[i+offset] = table[0][i];
           table[0] = t2;
        }

        // Set b to the square of the base
        int[] b = squareToLen(table[0], modLen, null);
        b = montReduce(b, mod, modLen, inv);

        // Set t to high half of b
        int[] t = new int[modLen];
        for(int i=0; i<modLen; i++)
            t[i] = b[i];

        // Fill in the table with odd powers of the base
        for (int i=1; i<tblmask; i++) {
            int[] prod = multiplyToLen(t, modLen, table[i-1], ...
               modLen, null);
            table[i] = montReduce(prod, mod, modLen, inv);
        }

        // Pre load the window that slides over the exponent
        int bitpos = 1 << ((ebits-1) & (32-1));

        int buf = 0;
        int elen = exp.length;
        int eIndex = 0;
        for (int i = 0; i <= wbits; i++) {
            buf = (buf << 1) | (((exp[eIndex] & bitpos) != 0)?1:0)...
               ;
            bitpos >>>= 1;
            if (bitpos == 0) {
                eIndex++;
                bitpos = 1 << (32-1);
                elen--;
            }
        }

        int multpos = ebits;

        // The first iteration, which is hoisted out of the main ...
           loop
        ebits--;
        boolean isone = true;

        multpos = ebits - wbits;
        while ((buf & 1) == 0) {
            buf >>>= 1;
            multpos++;
        }

        int[] mult = table[buf >>> 1];

        buf = 0;
        if (multpos == ebits)
            isone = false;

        // The main loop
        while(true) {
            ebits--;
            // Advance the window
            buf <<= 1;

            if (elen != 0) {
                buf |= ((exp[eIndex] & bitpos) != 0) ? 1 : 0;
                bitpos >>>= 1;
                if (bitpos == 0) {
                    eIndex++;
                    bitpos = 1 << (32-1);
                    elen--;
                }
            }

            // Examine the window for pending multiplies
            if ((buf & tblmask) != 0) {
                multpos = ebits - wbits;
                while ((buf & 1) == 0) {
                    buf >>>= 1;
                    multpos++;
                }
                mult = table[buf >>> 1];
                buf = 0;
            }

            // Perform multiply
            if (ebits == multpos) {
                if (isone) {
                    b = new int[mult.length];//mult.clone();
                                         for(int i = 0; i < mult....
                                            length; i++)
                                                 b[i] = mult[i];
                    isone = false;
                } else {
                    t = b;
                    a = multiplyToLen(t, modLen, mult, modLen, a);
                    a = montReduce(a, mod, modLen, inv);
                    t = a; a = b; b = t;
                }
            }

            // Check if done
            if (ebits == 0)
                break;

            // Square the input
            if (!isone) {
                t = b;
                a = squareToLen(t, modLen, a);
                a = montReduce(a, mod, modLen, inv);
                t = a; a = b; b = t;
            }
        }

        // Convert result out of Montgomery form and return
        int[] t2 = new int[2*modLen];
        for(int i=0; i<modLen; i++)
            t2[i+modLen] = b[i];

        b = montReduce(t2, mod, modLen, inv);

        t2 = new int[modLen];
        for(int i=0; i<modLen; i++)
            t2[i] = b[i];

        return new BigInteger(1, t2);
    }
```

Appendix B. Java Software Implementations

Listing B.1. DAILMOM

```
library ieee;
use ieee.std_logic_1164.all;
entity and2 is
   port(in1, in2 : in std_logic;
        out1 : out std_logic);
end and2;
architecture concurrent of and2 is
begin
   out1 <= in1 and in2;
end concurrent;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
entity telephone_booth is
        generic ( n : integer := 32 );
        port (b, upper_in, lower_in : in  std_logic_vector(n-1 ...
           downto 0);
                           upper_out, lower_out : out ...
                              std_logic_vector(n-1 downto 0);
                                          operation : in  ...
                                             std_logic_vector(  1 ...
                                             downto 0);
                                          reset : in std_logic;
                                          done : out std_logic);
end entity telephone_booth;

-- Behavioral model for an 8-bit Booth Multiplier component
architecture mult of telephone_booth is
signal upper : std_logic_vector(n-1 downto 0);
signal donev : std_logic := '0';
begin
        process(b, upper_in, lower_in, operation, reset)
        begin
          if (reset = '1') then
           donev <= '0';
          else
                  if    ( operation = "10" ) then -- from 0 to 1
                         upper <= upper_in - b; -- subtract 'b' ...
                            from partial
                  elsif ( operation = "01" ) then -- from 1 to 0
                          upper <= upper_in + b; -- add 'b' to ...
                             partial
                  else
                          upper <= upper_in;
                  end if; -- else operation = "00" or "11"
                  donev <= '1';
```

```
                end if;
        end process;
    lower_out <= upper(0)   & lower_in(n-1 downto 1); -- shift ...
       lower half right
    upper_out <= upper(n-1) &    upper(n-1 downto 1); -- shift ...
       upper half right
    done <= donev;
end architecture mult;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity booth_mult is
        generic ( n : integer := 32 );
        port (ain, bin : in  std_logic_vector((2*n)-1 downto 0);
                  prod : out std_logic_vector((4*n)-1 downto 0);
                  reset : in std_logic;
                  done: out std_logic );
end entity booth_mult;

-- Structural model for a 32-bit Booth Multiplier outputing a 64-...
   bit result
architecture mult of booth_mult is

component telephone_booth is
        generic ( n : integer := 32 );
        port (b, upper_in, lower_in : in  std_logic_vector(n-1 ...
           downto 0);
                   upper_out, lower_out : out std_logic_vector(n-1...
                       downto 0);
                                          operation : in  ...
                                             std_logic_vector(  1 ...
                                             downto 0);
                                          reset : in std_logic;
                                                                                  d
end component;

component and2
   port(in1, in2: in std_logic;
        out1: out std_logic);
```

```
end component;

  signal ain1, ain2, bin1, bin2 : std_logic_vector(n-1 downto 0);
  signal prod1, prod2 : std_logic_vector((2*n)-1 downto 0);

        signal oper : std_logic_vector(n downto 0) := (others => ...
           ’0’);
  signal im : std_logic_vector((n*((2*(n))+2)-1) downto 0) := (...
     others => ’0’);
  signal donev : std_logic_vector(n downto 0) := (others => ’0’);
  signal doneu : std_logic_vector(n downto 0) := (others => ’0’);

        signal oper2 : std_logic_vector(n downto 0) := (others => ...
           ’0’);
  signal im2 : std_logic_vector((n*((2*(n))+2)-1) downto 0) := (...
     others => ’0’);
  signal donev2 : std_logic_vector(n downto 0) := (others => ’0’);
  signal doneu2 : std_logic_vector(n downto 0) := (others => ’0’);

        signal reset_values : std_logic := ’0’;

        signal copy_bin1, copy_bin2, copy_ain1, copy_ain2 : ...
           std_logic_vector(n-1 downto 0) := (others => ’0’);

begin


make_in : FOR i IN 0 TO n-1 GENERATE
  ain2(i) <= ain(2*i);
  bin2(i) <= bin(2*i);
  ain1(i) <= ain((2*i)+1);
  bin1(i) <= bin((2*i)+1);
end generate;



    loopy    : FOR i IN 0 TO n-1 GENERATE
            tboot1ton : telephone_booth generic map(n => 32)
            port map ( b => bin1,
              upper_in => im(((n*((2*i)+1))-1) downto ((n*((2*i)...
                 +0))-0)),
              lower_in => im(((n*((2*i)+2))-1) downto ((n*((2*i)...
                 +1))-0)),
              upper_out => im((n*((2*(i+1))+1)-1) downto (n*((2*(i...
                 +1))+0)-0)),
              lower_out => im((n*((2*(i+1))+2)-1) downto (n*((2*(i...
                 +1))+1)-0)),
              operation => oper(i+1 downto i),
              reset => reset_values ,
              done => donev(i));

           END GENERATE;
    loopy2    : FOR i IN 0 TO n-1 GENERATE
            tboot1ton2 : telephone_booth generic map(n => 32)
            port map ( b => bin2,
              upper_in => im2(((n*((2*i)+1))-1) downto ((n*((2*i)...
                 +0))-0)),
              lower_in => im2(((n*((2*i)+2))-1) downto ((n*((2*i)...
                 +1))-0)),
              upper_out => im2((n*((2*(i+1))+1)-1) downto (n*((2*(...
                 i+1))+0)-0)),
              lower_out => im2((n*((2*(i+1))+2)-1) downto (n*((2*(...
                 i+1))+1)-0)),
              operation => oper2(i+1 downto i),
              reset => reset_values,
              done => donev2(i));

           END GENERATE;

doneu(0) <= '1';
doneu2(0) <= '1';

done_loopy : FOR i IN 0 TO n-1 GENERATE
  doneintermediate : and2 port map (in1 => doneu(i), in2 => donev(...
     i), out1 => doneu(i+1));
end generate;

done_loopy2 : FOR i IN 0 TO n-1 GENERATE
  doneintermediate2 : and2 port map (in1 => doneu2(i), in2 => ...
     donev2(i), out1 => doneu2(i+1));
end generate;

done <= doneu(n) and doneu2(n);

        process (bin2, ain1, ain2, reset) is
        begin

        if (reset = '1' or copy_ain1 /= ain1 or copy_ain2 /= ain2)...
            then
                im(n*(2)-1 downto n*(1)) <= ain1;
                im2(n*(2)-1 downto n*(1)) <= ain2;
                oper  <= ain1(n-1 downto 0) & '0';
                oper2  <= ain2(n-1 downto 0) & '0';
                reset_values <= '1';
                copy_ain1 <= ain1;
                copy_ain2 <= ain2;
        else
                reset_values <= '0';
                copy_bin1 <= bin1;
                copy_bin2 <= bin2;
        end if;

        end process;
```

```
--      process (bin1, bin2) is
--      begin
--              reset_values <= '0';
--      end process;

    prod1 <=    im((n*((2*(n))+1)-1) downto (n*((2*(n))+0)-0)) & ...
       im((n*((2*(n))+2)-1) downto (n*((2*(n))+1)-0));
    prod2 <=    im2((n*((2*(n))+1)-1) downto (n*((2*(n))+0)-0)) & ...
       im2((n*((2*(n))+2)-1) downto (n*((2*(n))+1)-0));


make_out_at_piano_bar : FOR i IN 0 TO ((2*n)-1) GENERATE
  prod(2*i) <= prod2(i);
  prod((2*i)+1) <= prod1(i);
end generate;



end architecture mult;
```

Listing B.2. Main RCA definition

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

USE WORK.ALL;

ENTITY add_8 IS
    PORT(
        x_in    :    IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        y_in    :    IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        c_in    :    IN STD_LOGIC;
        sum     :    OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        c_out   :    OUT STD_LOGIC);
END add_8;

ARCHITECTURE struct OF add_8 IS


component full_add is
  PORT(
          a      : IN STD_LOGIC;
          b      : IN STD_LOGIC;
          c_in   : IN STD_LOGIC;
          sum    : OUT STD_LOGIC;
          c_out  : OUT STD_LOGIC);
END component;

component carry is
  PORT(
      c_in : in std_logic;
      c_out : out std_logic);
```

```
end component;

SIGNAL im  :    STD_LOGIC_VECTOR(30 DOWNTO 0);
SIGNAL imi :    STD_LOGIC_VECTOR(30 DOWNTO 0);
BEGIN
    c0   : full_add
           PORT MAP (x_in(0),y_in(0),c_in,sum(0),im(0));
    c01  : carry
           PORT MAP (im(0),imi(0));
    c    : FOR i IN 1 TO 30 GENERATE
            c1to6:  full_add PORT MAP (x_in(i),y_in(i),
            imi(i-1),sum(i),im(i));
            c11to16: carry PORT MAP (im(i),imi(i));
           END GENERATE;
    c7   : full_add PORT MAP (x_in(31),y_in(31),
           imi(30),sum(31),c_out);
END struct;
```

Listing B.3. Full Adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

use work.all;

entity full_add is
      PORT(
          a      : IN STD_LOGIC;
          b      : IN STD_LOGIC;
          c_in   : IN STD_LOGIC;
          sum    : OUT STD_LOGIC;
          c_out  : OUT STD_LOGIC);
END full_add;

ARCHITECTURE behv OF full_add IS
BEGIN
    sum <= a XOR b XOR c_in;
    c_out <= (a AND b) OR (c_in AND (a OR b));
END behv;
```

Listing B.4. Carry

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

use work.all;

entity carry is
      PORT(
          c_in   : in STD_LOGIC;
          c_out  : out STD_LOGIC);
END carry;

```

```
ARCHITECTURE behv OF carry IS
signal carry : std_logic;

BEGIN
    carry <= c_in;
    c_out <= carry;

END behv;
```

Listing B.5. Main CLA Adder definition

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY c_l_addr IS
    PORT
        (
         x_in      :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         y_in      :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_in  :  IN   STD_LOGIC;
         sum       :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_out :  OUT  STD_LOGIC
        );
END c_l_addr;

ARCHITECTURE behavioral OF c_l_addr IS

SIGNAL    h_sum                :    STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    carry_generate       :    STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    carry_propagate      :    STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    carry_in_internal    :    STD_LOGIC_VECTOR(31 DOWNTO 1);

BEGIN
    h_sum <= x_in XOR y_in;
    carry_generate <= x_in AND y_in;
    carry_propagate <= x_in OR y_in;
    PROCESS (carry_generate,carry_propagate,carry_in_internal)
    BEGIN
    carry_in_internal(1) <= carry_generate(0) OR (carry_propagate...
       (0) AND carry_in);
        inst: FOR i IN 1 TO 30 LOOP
              carry_in_internal(i+1) <= carry_generate(i) OR (...
                 carry_propagate(i) AND carry_in_internal(i));
              END LOOP;
    carry_out <= carry_generate(31) OR (carry_propagate(31) AND ...
       carry_in_internal(31));
    END PROCESS;

    sum(0) <= h_sum(0) XOR carry_in;
    sum(31 DOWNTO 1) <= h_sum(31 DOWNTO 1) XOR carry_in_internal...
       (31 DOWNTO 1);
END behavioral;
```

Listing B.6. Math Driver Hardware

```
--
--
--  This file is a part of JOP, the Java Optimized Processor
--
--  Copyright (C) 2001-2008, Martin Schoeberl (martin@jopdesign....
    com)
--
--  This program is free software: you can redistribute it and/or ...
    modify
--  it under the terms of the GNU General Public License as ...
    published by
--  the Free Software Foundation, either version 3 of the License,...
     or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful...
    ,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public ...
    License
--  along with this program.  If not, see <http://www.gnu.org/...
    licenses/>.
--



--
--      sc_test_slave.vhd
--
--      A simple test slave for the SimpCon interface
--
--      Author: Martin Schoeberl        martin@jopdesign.com
--
--
--      resources on Cyclone
--
--              xx LCs, max xx MHz
--
--
--      2005-11-29      first version
--
--      todo:
--
--


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity sc_math is
generic (addr_bits : integer);

port (
        clk             : in std_logic;
        reset   : in std_logic;

-- SimpCon interface

        address         : in std_logic_vector(addr_bits-1 downto ...
           0);
        wr_data         : in std_logic_vector(31 downto 0);
        rd, wr          : in std_logic;
        rd_data         : out std_logic_vector(31 downto 0);
        rdy_cnt         : out unsigned(1 downto 0)

);
end sc_math;

architecture rtl of sc_math is

component c_l_addr IS
    PORT
        (
         x_in      :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         y_in      :  IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_in  :  IN   STD_LOGIC;
         sum       :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
         carry_out :  OUT  STD_LOGIC
        );
END component;

component add_8 IS
    PORT(
        x_in    :    IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        y_in    :    IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        c_in    :    IN STD_LOGIC;
        sum     :    OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        c_out   :    OUT STD_LOGIC);
END component;

component booth_mult is
        generic ( n : integer := 32 );
        port (ain, bin : in  std_logic_vector((2*n)-1 downto 0);
                  prod : out std_logic_vector((4*n)-1 downto 0);
                  reset : in std_logic;
                  done: out std_logic );
end component;


        signal sum1                    : std_logic_vector(31 ...
           downto 0);
        signal a1                       : std_logic_vector(31 ...
           downto 0); -- := "00000000000000000000000000011010";
        signal b1                       : std_logic_vector(31 ...
           downto 0); -- := "00000000000000000000000000001101";
        signal c_o1 : std_logic;

        signal sum2                     : std_logic_vector(31 ...
           downto 0);
        signal a2                       : std_logic_vector(31 ...
           downto 0); -- := "00000000000000000000000000011010";
        signal b2                       : std_logic_vector(31 ...
           downto 0); -- := "00000000000000000000000000001101";
        signal c_o2 : std_logic;

        signal ain, bin : std_logic_vector(63 downto 0);
        signal prod : std_logic_vector(127 downto 0);
        signal reset_booth : std_logic;
        signal done: std_logic ;

begin

ad : c_l_addr port map (a1,b1,'0',sum1,c_o1);
ad2 : add_8 port map (a2,b2,'0',sum2,c_o2);
bm : booth_mult port map (ain, bin, prod, reset_booth, done);

--
--      The registered MUX is all we need for a SimpCon read.
--      The read data is stored in registered rd_data.
--
process(clk, reset)
begin

        if (reset='1') then
                rd_data <= (others => '0');
        elsif rising_edge(clk) then

                if rd='1' then
                        -- that's our very simple address decoder
                        case address(3 downto 0) is
                                when "0000" =>
                                        rd_data <= sum1;
                                when "0001" =>
                                        rd_data <= sum2;
                                when "0010" =>
                                        rd_data <= prod(127 downto...
                                            96);
                                when "0011" =>
                                        rd_data <= prod(95 downto ...
                                           64);
                                when "0100" =>
                                        rd_data <= prod(63 downto ...
                                           32);
                                when "0101" =>
                                        rd_data <= prod(31 downto ...
                                           0);
                                when "0110" =>
                                        rd_data(31 downto 1) <= (...
                                           others => ’0’);
                                        rd_data(0) <= done;
                                when others =>
                                        rd_data(31 downto 0) <= (...
                                           others => ’0’);
                                end case;
                end if;
        end if;

end process;


--
--      SimpCon write is very simple
--
process(clk, reset)

begin

        if (reset=’1’) then
                a1 <= (others => ’0’);
                b1 <= (others => ’0’);
                a2 <= (others => ’0’);
                b2 <= (others => ’0’);
                ain <= (others => ’0’);
                bin <= (others => ’0’);

        elsif rising_edge(clk) then
                if wr=’1’ then
                        case address(3 downto 0) is
                                when "0000" =>
                                        a1 <= wr_data;
                                when "0001" =>
                                        b1 <= wr_data;
                                when "0010" =>
                                        a2 <= wr_data;
                                when "0011" =>
                                        b2 <= wr_data;
                                when "0100" =>
                                        ain(63 downto 32) <= ...
                                           wr_data;
                                when "0101" =>
                                        ain(31 downto 0) <= ...
                                           wr_data;
                                when "0110" =>
                                        bin(63 downto 32) <= ...
```

```
                                           wr_data;
                                when "0111" =>
                                        bin(31 downto 0) <= ...
                                           wr_data;
                                when others =>

                                end case;
                end if;

        end if;

end process;

end rtl;
```

Listing B.7. Assembler code override

```
//moved to jvm.java
//iadd:         add nxt

//moved to jvm.java
//imul:
//                      stmul           // store both operands and...
    start
//                      pop                     // pop second ...
   operand

//                      ldi     2               // 2*7+2 wait ok!
//imul_loop:
//                      ldi     -1
//                      add
//                      dup
//                      nop
//                      bnz     imul_loop
//                      nop
//                      nop

//                      pop                     // remove counter

//                      ldmul   nxt
```

Listing B.8. Assembler code override

```
        private static int add_op = 1;
        private static int f_iadd(int a, int b) {

                if (add_op == 1){
                        add_op = 2;
                        Native.wr(a, Const.IO_ADD_1_a);
                        Native.wr(b, Const.IO_ADD_1_b);
                        return Native.rd(Const.IO_ADD_1);
                        }
                else {
                        add_op = 1;
                        Native.wr(a, Const.IO_ADD_2_a);
                        Native.wr(b, Const.IO_ADD_2_b);
                        return Native.rd(Const.IO_ADD_2);
                        }

        }


        private static int mult_op = 1;
        private static int f_imul(int a, int b) {

        int confound_a;
        int confound_b;


                if (mult_op == 1) {
                        confound_a = 0xAAAA0000;
                        confound_b = 0xAAAAAAAA;
                        mult_op++;
                        }
                        else if (mult_op == 2) {
                        confound_a = 0x0000AAAA;
                        confound_b = 0xAAAAAAAA;
                        mult_op++;
                        }
                        else if (mult_op == 3) {
                        confound_a = 0xAA0000AA;
                        confound_b = 0xAAAAAAAA;
                        mult_op++;
                        }
                        else if (mult_op == 4) {
                        confound_a = 0x00AAAA00;
                        confound_b = 0xAAAAAAAA;
                        mult_op++;
                        }
                        else if (mult_op == 4) {
                        confound_a = 0x00000000;
                        confound_b = 0xAAAAAAAA;
                        mult_op++;
                        }
                        else {
                        confound_a = 0xAAAAAAAA;
                        confound_b = 0xAAAAAAAA;
                        mult_op = 1;
                        }
```

```
int a_temp = a >>16;
int b_temp = confound_a >>16;
int c_high = 0x0;


            for (int i = 31; i >= 0; i--){

                  if ((((i-1)>>1) == (i>>1))...
                     && (i != 0)){
                        c_high=c_high+((((...
                          a_temp>>(i>>1))...
                          &0x1))<<(i));
                  }
                  else{
                        c_high=c_high+((((...
                          b_temp>>(i>>1))...
                          &0x1))<<(i));
                  }
            }


            a_temp = (a & 0xffff);
            b_temp = (confound_a & 0xffff);
            int c_low = 0x0;


            for (int i = 31; i >= 0; i--){

                  if ((((i-1)>>1) == (i>>1))...
                     && (i != 0)){
                        c_low=c_low+((((...
                          a_temp>>(i>>1))...
                          &0x1))<<(i));
                  }
                  else{
                        c_low=c_low+((((...
                          b_temp>>(i>>1))...
                          &0x1))<<(i));
                  }
            }


a_temp = b >>16;
b_temp = confound_b >>16;
int d_high = 0x0;


            for (int i = 31; i >= 0; i--){

                  if ((((i-1)>>1) == (i>>1))...
                     && (i != 0)){
                        d_high=d_high+((((...
                                                   a_temp>>(i>>1))...
                                                   &0x1))<<(i));
                                        }
                                        else{
                                                d_high=d_high+((((...
                                                   b_temp>>(i>>1))...
                                                   &0x1))<<(i));
                                        }
                                }


                                a_temp = (b & 0xffff);
                                b_temp = (confound_b & 0xffff);
                                int d_low = 0x0;


                                for (int i = 31; i >= 0; i--){

                                        if ((((i-1)>>1) == (i>>1))...
                                            && (i != 0)){
                                                d_low=d_low+((((...
                                                   a_temp>>(i>>1))...
                                                   &0x1))<<(i));
                                        }
                                        else{
                                                d_low=d_low+((((...
                                                   b_temp>>(i>>1))...
                                                   &0x1))<<(i));
                                        }
                                }

                                Native.wr(c_high,Const.IO_MULT_a1)...
                                   ;
                                Native.wr(c_low,Const.IO_MULT_a2);
                                Native.wr(d_high,Const.IO_MULT_b1)...
                                   ;
                                Native.wr(d_low,Const.IO_MULT_b2);

                                        while(Native.rd(Const....
                                           IO_MULT_done)==0);

//                              int returned1 = Native.rd(Const....
   IO_MULT_1);
//                              int returned2 = Native.rd(Const....
   IO_MULT_2);
                                int returned3 = Native.rd(Const....
                                   IO_MULT_3);
                                int returned4 = Native.rd(Const....
                                   IO_MULT_4);

//                              int val_a_high = 0x0;
                                int val_a_low = 0x0;
```

```
//                              int val_b_high = 0x0;
//                              int val_b_low = 0x0;

//                              for (int i = 31; i >= 0; i--){
//                                      if (i>>1 == (i-1)>>1)
//                                              val_a_high = ...
   val_a_high + ((returned1>>(i)&0x1)<<((i>>1)+16));
//                                      else //if (i>>1 != (i-1)...
   >>1)
//                                              val_b_high = ...
   val_b_high + ((returned1>>(i)&0x1)<<((i>>1)+16));
//                              }

//                              for (int i = 31; i >= 0; i--){
//                                      if (i>>1 == (i-1)>>1)
//                                              val_a_high = ...
   val_a_high + ((returned2>>(i)&0x1)<<((i>>1)));
//                                      else
//                                              val_b_high = ...
   val_b_high + ((returned2>>(i)&0x1)<<((i>>1)));
//                              }


                                for (int i = 31; i >= 0; i--){
                                        if (i>>1 == (i-1)>>1)
                                                val_a_low = ...
                                                   val_a_low + ((...
                                                   returned3>>(i)...
                                                   &0x1)<<((i>>1)...
                                                   +16));
//                                      else //if (i>>1 != (i-1)...
   >>1)
//                                              val_b_low = ...
   val_b_low + ((returned3>>(i)&0x1)<<((i>>1)+16));
                                }

                                for (int i = 31; i >= 0; i--){
                                        if (i>>1 == (i-1)>>1)
                                                val_a_low = ...
                                                   val_a_low + ((...
                                                   returned4>>(i)...
                                                   &0x1)<<((i>>1))...
                                                   );
//                                      else
//                                              val_b_low = ...
   val_b_low + ((returned4>>(i)&0x1)<<((i>>1)));
                                }

        return val_a_low;

        }
```

Listing B.9. JOP Port Definitions

```
//addres
//read
public static final int IO_ADD_1 = IO_BASE + 0x50 + 0;
public static final int IO_ADD_2 = IO_BASE + 0x50 + 1;
public static final int IO_MULT_1 = IO_BASE + 0x50 + 2;
public static final int IO_MULT_2 = IO_BASE + 0x50 + 3;
public static final int IO_MULT_3 = IO_BASE + 0x50 + 4;
public static final int IO_MULT_4 = IO_BASE + 0x50 + 5;
public static final int IO_MULT_done = IO_BASE + 0x50 + 6;


//write
public static final int IO_ADD_1_a = IO_BASE + 0x50 + 0;
public static final int IO_ADD_1_b = IO_BASE + 0x50 + 1;
public static final int IO_ADD_2_a = IO_BASE + 0x50 + 2;
public static final int IO_ADD_2_b = IO_BASE + 0x50 + 3;
public static final int IO_MULT_a1 = IO_BASE + 0x50 + 4;
public static final int IO_MULT_a2 = IO_BASE + 0x50 + 5;
public static final int IO_MULT_b1 = IO_BASE + 0x50 + 6;
public static final int IO_MULT_b2 = IO_BASE + 0x50 + 7;
```

Appendix C. Java Software Implementations

Listing C.1. Main AES control class

```
package test;
import com.jopdesign.sys.Const;
import com.jopdesign.sys.Native;

public class AES_main_independant {

        static int S_Box[][] =             {
                {0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0...
                   x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76},
                {0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0...
                   xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0},
                {0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0...
                   x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15},
                {0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0...
                   x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75},
                {0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0...
                   x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84},
                {0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0...
                   x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf},
                {0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0...
                   x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8},
                {0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0...
                   xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2},
                {0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0...
                   xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73},
                {0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0...
                   x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb},
                {0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0...
                   xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79},
                {0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0...
                   x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08},
                {0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0...
                   xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a},
                {0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0...
                   x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e},
                {0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0...
                   x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf},
                {0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0...
                   x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16}
        };


        //byte[][] key_square = {{(byte)0x2b,(byte)0x7e,(byte)0x15...
           ,(byte)0x16},{(byte)0x28,(byte)0xae,(byte)0xd2,(byte)0...
           xa6},{(byte)0xab,(byte)0xf7,(byte)0x15,(byte)0x88},{(...
           byte)0x09,(byte)0xcf,(byte)0x4f,(byte)0x3c}};
        static byte[][] key_square = {{(byte)0x00,(byte)0x11,(byte...
           )0x22,(byte)0x33},{(byte)0x44,(byte)0x55,(byte)0x66,(...
```

```
      byte)0x77},{(byte)0x88,(byte)0x99,(byte)0xaa,(byte)0xbb...
      },{(byte)0xcc,(byte)0xdd,(byte)0xee,(byte)0xff}};

    byte[][] state = new byte[4][4];        //plaintext will ...
      copy to here
    static byte[][] mixCol = {{2,3,1,1}, {1,2,3,1}, {1,1,2,3},...
       {3,1,1,2}};

    byte[] result = new byte[16];

    byte[][] newState = new byte[4][4];
    byte[][] tempSt = new byte[4][4];
    byte temp;

    byte[][] row1 = new byte[11][4];
    byte[][] row2 = new byte[11][4];
    byte[][] row3 = new byte[11][4];
    byte[][] row4 = new byte[11][4];
    byte[][] current_keyex = new byte[4][4];



    public byte[] get_r_done(byte[] plain){

        for (int j = 0; j < 4; j++)
            for(int pop = 0; pop < 4; pop++)
                this.state[pop][j] = plain[(4*j)+...
                  pop];

//////////////////////////////////////////////

        row1[0] = key_square[0];
        row2[0] = key_square[1];
        row3[0] = key_square[2];
        row4[0] = key_square[3];

        byte[][] key1 = key_square;
        byte[][] key2 = new byte[4][4];

        byte[] rotword = new byte[4];
        byte[] recon = {0x01,0x02,0x04,0x08,0x10,0x20,0x40...
          ,(byte)0x80,0x1b,0x36};

        for (int i = 1; i< 11; i++){

        rotword[0] = (byte)S_Box[(key1[3][1]>>4)&0xf][key1...
          [3][1]&0xf];
        rotword[1] = (byte)S_Box[(key1[3][2]>>4)&0xf][key1...
          [3][2]&0xf];
        rotword[2] = (byte)S_Box[(key1[3][3]>>4)&0xf][key1...
          [3][3]&0xf];
```

```
                rotword[3] = (byte)S_Box[(key1[3][0]>>4)&0xf][key1...
                    [3][0]&0xf];

                key2[0][0] = (byte)(key1[0][0]^rotword[0]^recon[i...
                    -1]);
                key2[0][1] = (byte)(key1[0][1]^rotword[1]);
                key2[0][2] = (byte)(key1[0][2]^rotword[2]);
                key2[0][3] = (byte)(key1[0][3]^rotword[3]);

                key2[1][0] = (byte)(key1[1][0]^key2[0][0]);
                key2[1][1] = (byte)(key1[1][1]^key2[0][1]);
                key2[1][2] = (byte)(key1[1][2]^key2[0][2]);
                key2[1][3] = (byte)(key1[1][3]^key2[0][3]);

                key2[2][0] = (byte)(key1[2][0]^key2[1][0]);
                key2[2][1] = (byte)(key1[2][1]^key2[1][1]);
                key2[2][2] = (byte)(key1[2][2]^key2[1][2]);
                key2[2][3] = (byte)(key1[2][3]^key2[1][3]);

                key2[3][0] = (byte)(key1[3][0]^key2[2][0]);
                key2[3][1] = (byte)(key1[3][1]^key2[2][1]);
                key2[3][2] = (byte)(key1[3][2]^key2[2][2]);
                key2[3][3] = (byte)(key1[3][3]^key2[2][3]);


                System.arraycopy(key2[0], 0, row1[i], 0, 4);
                System.arraycopy(key2[1], 0, row2[i], 0, 4);
                System.arraycopy(key2[2], 0, row3[i], 0, 4);
                System.arraycopy(key2[3], 0, row4[i], 0, 4);

                key1 = key2;

                }

////////////////////////////////////////////////



//              this.addRoundKey(this.state, 0);
                current_keyex[0] = row1[0];
                current_keyex[1] = row2[0];
                current_keyex[2] = row3[0];
                current_keyex[3] = row4[0];

                for(int j = 0; j<4; j++){
                        for(int i = 0; i<4; i++){
                                state[i][j] = (byte) (...
                                    current_keyex[j][i] ^ state[i][...
                                    j]);
                        }
                }
                Native.wr(0x01, Const.IO_LED5);

                for(int round = 1; round < 10; round++){

//              this.state = this.subBytes(this.state);
                        for (int j = 0; j < 4; j++)
                                for(int dog = 0; dog < 4; dog++)
                                        state[dog][j] = (byte)...
                                           S_Box[(state[dog][j...
                                           ]>>4)&0xf][state[dog][j...
                                           ]&0xf];




//              this.shiftRows(this.state);
                temp = state[1][0];
                for(int star = 0; star<3; star++){
                        state[1][star] = state[1][star+1];
                }
                state[1][3] = temp;
                temp = state[2][0];
                byte temp1 = state[2][1];
                for(int moon = 0; moon<2; moon++){
                        state[2][moon] = state[2][moon+2];
                }
                state[2][3] = temp1;
                state[2][2] = temp;
                temp = state[3][0];
                temp1 = state[3][1];
                byte temp2 = state[3][2];
                byte temp3 = state[3][3];
                state[3][0] = temp3;
                state[3][1] = temp;
                state[3][2] = temp1;
                state[3][3] = temp2;




//              this.mixColumns(this.state);
                for(int red = 0; red<4; red++){

                        tempSt[0][red] = (byte) ((this.shifty(...
                           state[0][red], 2))^(this.shifty(state...
                           [1][red], 3))^state[2][red]^state[3][...
                           red]);
                        tempSt[1][red] = (byte) (state[0][red]^...
                           this.shifty(state[1][red], 2)^this....
                           shifty(state[2][red], 3)^ state[3][red...
                           ]);
                        tempSt[2][red] = (byte) (state[0][red]^...
                           state[1][red]^this.shifty(state[2][red...
                           ],2)^this.shifty(state[3][red], 3));
                        tempSt[3][red] = (byte) (this.shifty(state...
                           [0][red], 3)^state[1][red]^state[2][red...
                           ]^this.shifty(state[3][red], 2));
                }

                System.arraycopy(tempSt[0], 0, state[0], 0, 4);
                System.arraycopy(tempSt[1], 0, state[1], 0, 4);
                System.arraycopy(tempSt[2], 0, state[2], 0, 4);
                System.arraycopy(tempSt[3], 0, state[3], 0, 4);


                //              this.addRoundKey(this.state, round...
                   );
                current_keyex[0] = row1[round];
                current_keyex[1] = row2[round];
                current_keyex[2] = row3[round];
                current_keyex[3] = row4[round];


                for(int j = 0; j<4; j++)
                        for(int i = 0; i<4; i++)
                                state[i][j] = (byte) (...
                                   current_keyex[j][i] ^ state[i][...
                                   j]);


                }

                if (true)
                        Native.wr(0x00, Const.IO_LED5);




//              this.state = this.subBytes(this.state);
                for (int j = 0; j < 4; j++)
                        for(int dog = 0; dog < 4; dog++)
                                state[dog][j] = (byte)S_Box[(state...
                                   [dog][j]>>4)&0xf][state[dog][j...
                                   ]&0xf];


//              this.shiftRows(this.state);
                temp = state[1][0];
                for(int star = 0; star<3; star++){
                        state[1][star] = state[1][star+1];
                }
                state[1][3] = temp;
                temp = state[2][0];
                byte temp1 = state[2][1];
                for(int moon = 0; moon<2; moon++){
                        state[2][moon] = state[2][moon+2];
                }
                state[2][3] = temp1;
                state[2][2] = temp;
                temp = state[3][0];
                temp1 = state[3][1];
                byte temp2 = state[3][2];
                byte temp3 = state[3][3];
                state[3][0] = temp3;
                state[3][1] = temp;
                state[3][2] = temp1;
                state[3][3] = temp2;



//              this.addRoundKey(this.state, 0);
                current_keyex[0] = row1[10];
                current_keyex[1] = row2[10];
                current_keyex[2] = row3[10];
                current_keyex[3] = row4[10];
                for(int j = 0; j<4; j++)
                        for(int i = 0; i<4; i++)
                                state[i][j] = (byte) (...
                                   current_keyex[j][i] ^ state[i][...
                                   j]);



//              return this.return_state();

                for(int j = 0; j<4; j++)
                        for(int pop = 0; pop < 4; pop++)
                                result[(j*4)+pop] = this.state[pop...
                                   ][j];


                return result;


        }


        public byte shifty(byte a, int b){
                byte mod = 27;
                if(b==2){
                        if(a < 0){
                                a = (byte) (a << 1);
                                a = (byte) (a^mod);
                        }
```

```
                    else a = (byte)(a<<1);
        }

        else if(b==3){
                    byte tempA = a;
                    if(a<0){
                                a = (byte) (a << 1);
                                a = (byte) (a^mod);
                                a = (byte)(a^tempA);
                    }
                    else {
                                a = (byte)((a<<1)^tempA);
                    }
        }
        return a;

        }













}
```

Appendix D. Double Stack

Listing D.1. Double Stack

```
--
-- xv4ram_block.vhd
--
-- Generated by BlockGen
-- Mon Jan 17 12:50:45 EST 2011
--
-- This module will synthesize on Spartan3 and Virtex2/2Pro/2ProX ...
   devices.
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
library unisim;
use unisim.vcomponents.all;

entity xram_block_double_ram is
        port (
                a_rst  : in std_logic;
                a_clk  : in std_logic;
                a_en   : in std_logic;
                a_wr   : in std_logic;
                a_addr : in std_logic_vector(8 downto 0);
                a_din1  : in std_logic_vector(31 downto 0);
                a_din2  : in std_logic_vector(31 downto 0);
                a_dout1 : out std_logic_vector(31 downto 0);
                a_dout2 : out std_logic_vector(31 downto 0);
                b_rst  : in std_logic;
                b_clk  : in std_logic;
                b_en   : in std_logic;
                b_wr   : in std_logic;
                b_addr : in std_logic_vector(8 downto 0);
                b_din1  : in std_logic_vector(31 downto 0);
                b_din2  : in std_logic_vector(31 downto 0);
                b_dout1 : out std_logic_vector(31 downto 0);
                b_dout2 : out std_logic_vector(31 downto 0)
        );
end xram_block_double_ram;

architecture rtl of xram_block_double_ram is

        component RAMB16_S36_S36
                port (
                        DIA    : in std_logic_vector (31 downto 0)...
                           ;
                        DIB    : in std_logic_vector (31 downto 0)...
                           ;
                        ENA    : in std_logic;
                        ENB    : in std_logic;
                        WEA    : in std_logic;
                        WEB    : in std_logic;
                        SSRA   : in std_logic;
                        SSRB   : in std_logic;
                        DIPA   : in std_logic_vector (3 downto 0);
                        DIPB   : in std_logic_vector (3 downto 0);
                        DOPA   : out std_logic_vector (3 downto 0)...
                           ;
                        DOPB   : out std_logic_vector (3 downto 0)...
                           ;
                        CLKA   : in std_logic;
                        CLKB   : in std_logic;
                        ADDRA  : in std_logic_vector (8 downto 0);
                        ADDRB  : in std_logic_vector (8 downto 0);
                        DOA    : out std_logic_vector (31 downto ...
                           0);
                        DOB    : out std_logic_vector (31 downto ...
                           0)
                );
        end component;

        attribute INIT: string;
        attribute INIT_00: string;
        attribute INIT_01: string;
        attribute INIT_02: string;
        attribute INIT_03: string;
        attribute INIT_04: string;
        attribute INIT_05: string;
        attribute INIT_06: string;
        attribute INIT_07: string;
        attribute INIT_08: string;
        attribute INIT_09: string;
        attribute INIT_0a: string;
        attribute INIT_0b: string;
        attribute INIT_0c: string;
        attribute INIT_0d: string;
        attribute INIT_0e: string;
        attribute INIT_0f: string;
        attribute INIT_10: string;
        attribute INIT_11: string;
        attribute INIT_12: string;
        attribute INIT_13: string;
        attribute INIT_14: string;
        attribute INIT_15: string;
        attribute INIT_16: string;
        attribute INIT_17: string;
        attribute INIT_18: string;
        attribute INIT_19: string;
        attribute INIT_1a: string;
        attribute INIT_1b: string;
```

```
        attribute INIT_1c: string;
        attribute INIT_1d: string;
        attribute INIT_1e: string;
        attribute INIT_1f: string;
        attribute INIT_20: string;
        attribute INIT_21: string;
        attribute INIT_22: string;
        attribute INIT_23: string;
        attribute INIT_24: string;
        attribute INIT_25: string;
        attribute INIT_26: string;
        attribute INIT_27: string;
        attribute INIT_28: string;
        attribute INIT_29: string;
        attribute INIT_2a: string;
        attribute INIT_2b: string;
        attribute INIT_2c: string;
        attribute INIT_2d: string;
        attribute INIT_2e: string;
        attribute INIT_2f: string;
        attribute INIT_30: string;
        attribute INIT_31: string;
        attribute INIT_32: string;
        attribute INIT_33: string;
        attribute INIT_34: string;
        attribute INIT_35: string;
        attribute INIT_36: string;
        attribute INIT_37: string;
        attribute INIT_38: string;
        attribute INIT_39: string;
        attribute INIT_3a: string;
        attribute INIT_3b: string;
        attribute INIT_3c: string;
        attribute INIT_3d: string;
        attribute INIT_3e: string;
        attribute INIT_3f: string;

        attribute INIT_00 of cmp_ram_0: label is "...
           0000000000000000000000000000000000000000000000000000000000000000...
           ";
        attribute INIT_01 of cmp_ram_0: label is "...
           0000000000000000000000000000000000000000000000000000000000000000...
           ";
        attribute INIT_02 of cmp_ram_0: label is "...
           1234567800000000000000000000000000000000000000000000000000000000...
           ";
        attribute INIT_03 of cmp_ram_0: label is "...
           1234567812345678123456781234567812345678123456781234567812345678...
           ";
        attribute INIT_04 of cmp_ram_0: label is "...
           ffffff9100000002ffffff900000000400000000ffffff87ffffff8600000040...
           ";
```

```
attribute INIT_05 of cmp_ram_0: label is "0000001...
   fffffff840000000500000003000000060000 00ff0000000100000008...
   ";
attribute INIT_06 of cmp_ram_0: label is "...
   ffffff8f0000002000000003f80000000ffffff85ffffff800000ffffffffffff...
   ";
attribute INIT_07 of cmp_ram_0: label is "000000000132...
   b548123456781234567812345678123456781234567812345678";
attribute INIT_08 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_09 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0a of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0b of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0c of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0d of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0e of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0f of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_10 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_11 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_12 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_13 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_14 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_15 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_16 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_17 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_18 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_19 of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1a of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1b of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1c of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1d of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1e of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1f of cmp_ram_0: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_20 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_21 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_22 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_23 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_24 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_25 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_26 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_27 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
```

```
attribute INIT_28 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_29 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2a of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2b of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2c of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2d of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2e of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2f of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_30 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_31 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_32 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_33 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_34 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_35 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_36 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_37 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_38 of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_39 of cmp_ram_0: label is "...
```

```
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_3a of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_3b of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_3c of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_3d of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_3e of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_3f of cmp_ram_0: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";

        attribute INIT_00 of cmp_ram_1: label is "...
           0000000000000000000000000000000000000000000000000000000000000000
           ";
attribute INIT_01 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_02 of cmp_ram_1: label is "...
   1234567800000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_03 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_04 of cmp_ram_1: label is "...
   ffffff9100000002ffffff900000000400000000ffffff87ffffff8600000040...
   ";
attribute INIT_05 of cmp_ram_1: label is "0000001...
   fffffff840000000500000003000000060000000ff0000000100000008...
   ";
attribute INIT_06 of cmp_ram_1: label is "...
   ffffff8f0000000200000003f8000000ffffff85ffffff800000ffffffffffff...
   ";
attribute INIT_07 of cmp_ram_1: label is "000000000132...
   b548123456781234567812345678123456781234567812345678";
attribute INIT_08 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_09 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0a of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0b of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0c of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0d of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0e of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_0f of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_10 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_11 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_12 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_13 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_14 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_15 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_16 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_17 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_18 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_19 of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1a of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1b of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
```

```
attribute INIT_1c of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1d of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1e of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_1f of cmp_ram_1: label is "...
   1234567812345678123456781234567812345678123456781234567812345678...
   ";
attribute INIT_20 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_21 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_22 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_23 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_24 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_25 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_26 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_27 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_28 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_29 of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2a of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2b of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2c of cmp_ram_1: label is "...
   0000000000000000000000000000000000000000000000000000000000000000...
   ";
attribute INIT_2d of cmp_ram_1: label is "...
```

```
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_2e of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_2f of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_30 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_31 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_32 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_33 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_34 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_35 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_36 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_37 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_38 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_39 of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_3a of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_3b of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_3c of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_3d of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
       ";
    attribute INIT_3e of cmp_ram_1: label is "...
       0000000000000000000000000000000000000000000000000000000000000000...
```

```
           ";
        attribute INIT_3f of cmp_ram_1: label is "...
           0000000000000000000000000000000000000000000000000000000000000000...
           ";


        --signal p_a_addr : std_logic_vector (8 downto 0);
        --signal p_b_addr : std_logic_vector (8 downto 0);

        signal a_din_1  : std_logic_vector (31 downto 0);
        signal b_din_1  : std_logic_vector (31 downto 0);
        signal a_dout_1  : std_logic_vector (31 downto 0);
        signal b_dout_1  : std_logic_vector (31 downto 0);
        signal a_din_2  : std_logic_vector (31 downto 0);
        signal b_din_2  : std_logic_vector (31 downto 0);
        signal a_dout_2  : std_logic_vector (31 downto 0);
        signal b_dout_2  : std_logic_vector (31 downto 0);

        signal a : std_logic_vector (63 downto 0);

        signal b_all : std_logic_vector (63 downto 0);
        signal b : std_logic_vector (31 downto 0);
        signal b_not : std_logic_vector (31 downto 0);


begin


--    looploop    : FOR i IN 0 TO 15 GENERATE
--            b_dout(2*i) <= b_dout_1(2*i);
--                      b_dout(2*i+1) <= b_dout_2(2*i+1);
--
--                      a_din_1(2*i) <= a_din(2*i);
--                      a_din_1(2*i+1) <= not a_din(2*i);
--                      a_din_2(2*i) <= not a_din(2*i+1);
--                      a_din_2(2*i+1) <= a_din(2*i+1);
--
--           END GENERATE;



        --p_a_addr <= a_addr;
        --p_b_addr <= b_addr;

        cmp_ram_0 : RAMB16_S36_S36
                port map (
                        WEA => a_wr,
                        WEB => b_wr,
                        ENA => a_en,
                        ENB => b_en,
                        SSRA => a_rst,
                        SSRB => b_rst,
                        DIPA => "0000",
                        DIPB => "0000",
                        DOPA => open,
                        DOPB => open,
                        CLKA => a_clk,
                        CLKB => b_clk,
                        DIA => a_din1(31 downto 0),
                        ADDRA => a_addr,
                        DOA => a_dout1(31 downto 0),
                        DIB => b_din1(31 downto 0),
                        ADDRB => b_addr,
                        DOB => b_dout1(31 downto 0)
                );

        cmp_ram_1 : RAMB16_S36_S36
                port map (
                        WEA => a_wr,
                        WEB => b_wr,
                        ENA => a_en,
                        ENB => b_en,
                        SSRA => a_rst,
                        SSRB => b_rst,
                        DIPA => "0000",
                        DIPB => "0000",
                        DOPA => open,
                        DOPB => open,
                        CLKA => a_clk,
                        CLKB => b_clk,
                        DIA => a_din2(31 downto 0),
                        ADDRA => a_addr,
                        DOA => a_dout2(31 downto 0),
                        DIB => b_din2(31 downto 0),
                        ADDRB => b_addr,
                        DOB => b_dout2(31 downto 0)
                );


end rtl;
```

Appendix E. Double RAM

Listing E.1. sc_sram32.vhd

```
--
--
--  This file is a part of JOP, the Java Optimized Processor
--
--  Copyright (C) 2001-2008, Martin Schoeberl (martin@jopdesign....
   com)
--
--  This program is free software: you can redistribute it and/or ...
   modify
--  it under the terms of the GNU General Public License as ...
   published by
--  the Free Software Foundation, either version 3 of the License,...
    or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful...
   ,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public ...
   License
--  along with this program.  If not, see <http://www.gnu.org/...
   licenses/>.
--


--
--      sc_sram32.vhd
--
--      SimpCon compliant external memory interface
--      for 32-bit SRAM (e.g. Cyclone board, Spartan-3 Starter Kit...
   )
--
--      Connection between mem_sc and the external memory bus
--
--      memory mapping
--
--              000000-x7ffff   external SRAM (w mirror)        ...
   max. 512 kW (4*4 MBit)
--
--      RAM: 32 bit word
--
--
--      2005-11-22      first version
--      2007-03-17      changed SimpCon to records
--      2008-05-29      nwe on pos edge, additional wait state for...
    write
```

```
--

Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

use work.jop_types.all;
use work.sc_pack.all;

entity sc_mem_if is
generic (ram_ws : integer; addr_bits : integer);

port (

        clk, reset      : in std_logic;

--
--      SimpCon memory interface
--
        sc_mem_out              : in sc_out_type;
        sc_mem_in               : out sc_in_type;

-- memory interface

        ram_addr        : out std_logic_vector(addr_bits-1 downto ...
           0);
        ram_dout        : out std_logic_vector(31 downto 0);
        ram_din         : in std_logic_vector(31 downto 0);
        ram_dout_en     : out std_logic;
        ram_ncs         : out std_logic;
        ram_noe         : out std_logic;
        ram_nwe         : out std_logic

);
end sc_mem_if;

architecture rtl of sc_mem_if is

--
--      signals for mem interface
--
        type state_type         is (
                                                        idl, rd0a,...
                                                            rd1a, ...
                                                           rd2a, ...
                                                           rd0b, ...
                                                           rd1b, ...
                                                           rd2b,
                                                        wr0a, wr1a...
                                                           , wr2a,...
                                                            wridl,...
                                                            wr0b, ...
                                                            wr1b, ...
                                                            wr2b
                                                ) ;
        signal state                : state_type;
        signal next_state           : state_type;

        signal wait_state           : unsigned(4 downto 0);
        signal cnt                          : unsigned(1 downto 0);

        signal dout_ena             : std_logic;
        signal rd_data_ena          : std_logic;

        signal ram_ws_wr            : integer;

        signal ram_addr_intermediate        : std_logic_vector(...
           addr_bits-2 downto 0);
        signal ram_addr_0           : std_logic_vector(addr_bits-1 ...
           downto 0);
        signal ram_addr_1           : std_logic_vector(addr_bits-1 ...
           downto 0);

        signal ram_dout_buffer      : std_logic_vector(31 downto 0);
        signal ram_din_buffer       : std_logic_vector(31 downto 0);

        signal sram_data_in         : std_logic_vector(31 downto 0);
        signal sram_data_in_0       : std_logic_vector(31 downto 0);
        signal sram_data_in_1       : std_logic_vector(31 downto 0);
        signal sram_data_out        : std_logic_vector(31 downto 0);
        signal sram_data_out_0      : std_logic_vector(31 downto 0);
        signal sram_data_out_1      : std_logic_vector(31 downto 0);


begin

        ram_ws_wr <= ram_ws+1; -- additional wait state for SRAM

        assert SC_ADDR_SIZE>=addr_bits report "Too less address ...
           bits";
        ram_dout_en <= dout_ena;

        -- just keep it selected
        ram_ncs <= '0';

        sc_mem_in.rdy_cnt <= cnt;

--
--      Register memory address, write data and read data
--
process(clk, reset)
begin
        if reset='1' then
        end if;
end process;

--
--      next state logic
--
process(state, sc_mem_out.rd, sc_mem_out.wr, wait_state)

begin

        next_state <= state;


        case state is

                when idl =>

                        if sc_mem_out.rd='1' then
                                next_state <= rd0a;
                        elsif sc_mem_out.wr='1' then
                                next_state <= wr0a;
                        end if;


                when rd0a =>
                        if wait_state=8 then
                                next_state <= rd1a;
                        end if;
                when rd1a =>
                        if wait_state=6 then
                                next_state <= rd2a;
                        end if;

                -- last read state
                when rd2a =>
                        if wait_state=5 then
                                next_state <= rd0b;
                        end if;
                when rd0b =>
                        if wait_state=4 then
                                next_state <= rd1b;
                        end if;
                when rd1b =>
                        if wait_state=2 then
                                next_state <= rd2b;
                        end if;

                -- last read state
                when rd2b =>

                        next_state <= idl;
                        if sc_mem_out.rd='1' then
                                next_state <= rd0a;
                        elsif sc_mem_out.wr='1' then
                                next_state <= wr0a;
                        end if;

                -- the WS state
                when wr0a =>
                        if wait_state=13 then
                                next_state <= wr1a;
                        end if;
                when wr1a =>
                        if wait_state=10 then
                                next_state <= wr2a;
                        end if;

                -- last write state
                when wr2a =>
                        if wait_state=9 then
                                next_state <= wridl;
                        end if;

                when wridl =>
                        if wait_state=6 then
                                next_state <= wr0b;
                        end if;

                when wr0b =>
                        if wait_state=5 then
                                next_state <= wr1b;
                        end if;
                when wr1b =>
                        if wait_state=2 then
                                next_state <= wr2b;
                        end if;

                -- last write state
                when wr2b =>
                        if wait_state=1 then
                                next_state <= idl;
                        end if;

        end case;

end process;




--
--      state machine register
--      output register
--
process(clk, reset)

begin
        if (reset='1') then
                state <= idl;
                dout_ena <= '0';
                ram_noe <= '1';
                rd_data_ena <= '0';
                ram_nwe <= '1';

--...
   -----------------------------------------------------------------------------...


                ram_addr_intermediate <= (others => '0');
                ram_dout <= (others => '0');
                sc_mem_in.rd_data <= (others => '0');

        elsif rising_edge(clk) then

                if sc_mem_out.rd='1' or sc_mem_out.wr='1' then
                        ram_addr_intermediate <= sc_mem_out....
                           address(addr_bits-2 downto 0);
                end if;
                if sc_mem_out.wr='1' then
                        ram_dout_buffer <= sc_mem_out.wr_data;
                end if;
                if rd_data_ena='1' then
                        sc_mem_in.rd_data <= ram_din_buffer;
                end if;

--...
   --------------------------------------------------------------------------------



                state <= next_state;
                dout_ena <= '0';
                ram_noe <= '1';
                rd_data_ena <= '0';
                ram_nwe <= '1';

                case next_state is

                        when idl =>

                        -- the wait state
                        when rd0a =>
                                ram_addr <= ram_addr_0;
                        when rd1a =>
                                ram_noe <= ’0’;

                        -- last read state
                        when rd2a =>
                                sram_data_in_0 <= ram_din;
                                ram_noe <= ’0’;
--                              rd_data_ena <= ’1’;

                        when rd0b =>
                                ram_addr <= ram_addr_1;
                                ram_noe <= ’0’;

                        when rd1b =>
                                ram_noe <= ’0’;

                        -- last read state
                        when rd2b =>
                                sram_data_in_1 <= ram_din;

                                ram_noe <= ’0’;
                                rd_data_ena <= ’1’;

                        when wr0a =>
                                ram_addr <= ram_addr_0;
                                ram_dout <= sram_data_out_0;

                        when wr1a =>
                                ram_nwe <= ’0’;
                                dout_ena <= ’1’;

                        -- last write state
                        when wr2a =>
                                dout_ena <= ’1’;

                        when wridl =>

                        when wr0b =>
                                ram_addr <= ram_addr_1;
                                ram_dout <= sram_data_out_1;

                        when wr1b =>
                                ram_nwe <= ’0’;
                                dout_ena <= ’1’;

                        -- last write state
                        when wr2b =>
                                dout_ena <= ’1’;

                end case;

        end if;
end process;


           loopploop     : FOR i IN 0 TO 15 GENERATE
                        ram_din_buffer(2*i) <= sram_data_in_0(2*i)...
                           ;
                        ram_din_buffer(2*i+1) <= sram_data_in_1(2*...
                           i+1);

                        sram_data_out_0(2*i) <= ram_dout_buffer(2*...
                           i);
                        sram_data_out_0(2*i+1) <= not ...
                           ram_dout_buffer(2*i);
                        sram_data_out_1(2*i) <= not ...
                           ram_dout_buffer(2*i+1);
                        sram_data_out_1(2*i+1) <= ram_dout_buffer...
                           (2*i+1);

          END GENERATE;


--sram_data_out_0 <= not ram_dout_buffer;
--sram_data_out_1 <= ram_dout_buffer;
ram_addr_0 <= '0'&ram_addr_intermediate;
ram_addr_1 <= '1'&ram_addr_intermediate;
--ram_din_buffer <= sram_data_in_1;




--
-- wait_state processing
-- cs delay, dout enable
--
process(clk, reset)
begin
        if (reset='1') then
                wait_state <= (others => '1');
                cnt <= "00";
        elsif rising_edge(clk) then

                wait_state <= wait_state-1;

                cnt <= "11";
                if next_state=idl then
                        cnt <= "00";
                -- if wait_state<4 then
                elsif wait_state(4 downto 2)="000" then
                        cnt <= wait_state(1 downto 0)-1;
                end if;

                if sc_mem_out.rd='1' then
```

```
                        wait_state <= to_unsigned(ram_ws+8, 5);
                        if ram_ws<3 then
                                cnt <= to_unsigned(ram_ws+1, 2);
                        else
                                cnt <= "11";
                        end if;
                end if;

                if sc_mem_out.wr='1' then
                        wait_state <= to_unsigned(ram_ws_wr+11, 5)...
                            ;
                        if ram_ws_wr<3 then
                                cnt <= to_unsigned(ram_ws_wr+1, 2)...
                                    ;
                        else
                                cnt <= "11";
                        end if;
                end if;

        end if;
end process;

end rtl;
```

Bibliography

[1] "Tempest Standards". http://www.sst.ws/tempest_standards.php?pab=1_1, 2010.

[2] "Announcing the Advanced Encryption Standard (AES)". *Federal Information Processing Standards Publication 197*, November 26, 2001.

[3] Alam, S; Mohan MJ; Mukhopadhyay; Chowdhury DR; Gupta IS, M; Ghosh. "Effect of Glitches against Masked AES S-box Implementation and Countermeasure", 1 Oct, 2008.

[4] Anderson, Mike; Clulow Jolyon; Skorobogatov Sergei, Ross; Bond. "Cryptographic Processors - A Survey". *Cryptographic Processors - A Survey*, 94(2), February 2006.

[5] Anderson, Ross. "Why Cryptosystems Fail".

[6] Aumonier, Sebastien. "Generalized Correlation Power Analysis". *Oberthur Card Systems SA*.

[7] Chari, CS; Rao JR; Rohatgi P, S; Jutla. "Towards sound approaches to counteract power-analysis attacks". *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99) Springer-Verlag*, 398–412, 1999.

[8] Clavier, N, JCC; Dabbous. "Differential power analysis in the presence of hardware countermeasures". *Lectures Notes in Computer Science*, 1965:252–263, 2000.

[9] Goubin, J, L; Patarin. "DES and Differential Power Analysis (The "Duplication" Method)". *Cryptographic Hardware and Embedded Systems*, 158–172, 1999.

[10] J. Dyer, S.W. Smith, R. Perez and M. Lindemann. "Application support for a high-performance, programmable secure coprocessor". presented at the 22nd nat. Information Systems Security Conf., Arlington, VA, 1999.

[11] Kim, Takeshi; Homma Naofumi; Aoki Takafumi Satoh Akashi, Yongdae; Sugawara. "Biasing Power Traces to Improve Correlation in Power Analysis Attacks". *ICECE Technology, 2008. FPT 2008. International Conference on*, Dec 2008.

[12] Kim, Y.C., E.D. Trias, and D.R. Slaman. "Side channel analysis countermeasures using obfuscated instructions". *Security Technology (ICCST), 2010 IEEE International Carnahan Conference on*, 42 –51. 2010. ISSN 1071-6572.

[13] Kocher, J; Jun B, P; Jaffe. "Introduction to differential Power Analysis and Related Attacks". http://www.cryptography.com/dpa/technical, 1998.

[14] Kocher, J; Jun B, P; Jaffe. "Differential Power Analysis". *Proc. Advances in Cryptography*, 388–397, 1999.

[15] Kocher, J; Jun B, P; Jaffe. "Differential Power Analysis". *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[16] Kulikowski, MB; Taubin A, KJ; Karpovsky. "Power Attacks on Secure hardware Based on Early Propagation of Data". *IOLTS '06: Proceedings of the 12th IEEE International Symposium on On-Line Testing. IEEE Computer Society*, 131–138, 2006.

[17] Lu, Maire P.; McCanny John V., Yingxi; ONeill. "FPGA Implementation and Analysis of Random Delay Insertion Countermeasure against DPA". *ICECE Technology, 2008. FPT 2008. International Conference on*, Dec 2008.

[18] Mangard, T; Gammel BM, S; Popp. "Side Channel Leakage of Masked CMOS Gates". *The proceedings of CT-RSA*, 3376:351–365, 2005.

[19] Messerges, Ezzat; Sloan Robert, Thomas; Dabbish. "Examining Smart-Card Security under the Threat of Power analysis Attacks". *IEEE Transactions on Computers*, 51(5), May 2002.

[20] Messerges, Ezzy; Sloan Robert, Thomas; Dabbish. "Investigations of Power Analysis Attacks on Smartcards", 1999.

[21] Nohl, David; Plotz Starbug; Plotz Henryk, Karsten; Evans. "Reverse-Engineering a Cryptographic RFID Tag". *USENIX Security Symposium*, Jul 2008.

[22] Popp, S, T; Mangard. "masked Dual-Rail Pre-Charge Logic: DPA-resistance without Routing Constraints".

[23] Prouff, Matthiew; Bevan Regis, Emmanuel; Rivain. "Statistical Analysis of Second Order Differential Power Analysis". *Transactions on Computers*, 58(6), June 2009.

[24] Rammohan, S. "Reduced Complementary Dynamic and Differential Logic: a Circuit Design Methodology for DPA-resistant Cryptographic ICs". *Master's Thesis, University of Cincinnati*, May 2007.

[25] RISCure. "RSA Attacks". *RISCure Training Slides*, Mar 2010.

[26] Schaumont, K, P; Tiri. "Masking and Dual-Rail Logic dont Add Up". *9th Intl. Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)*, 95–106, 2007.

[27] Schoeberl, Martin. “Java Optimized Processor”. *http://www.jopdesign.com/.*

[28] Sundaresan, Srividhya; Vermuri Ranga, Vijay; Rammohan. “Defense against Side-Channel Power Analysis Attacks on Microelectronics Systems”, 2008.

[29] Suzuki, D. “Random Switching logic: A New Countermeasure against DPA and Second-Order DPA at the logic level”. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E90-A(1):160–168, 2007.

[30] Thomas Popp, Elisabeth Oswald, Stefan Mangard. “Power Analysis Attacks and Countermeasures”. *IEEE Design and Test of Computers*, 535–543, 2007.

[31] Tiri, K. “A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards”. *Proc. of European Solid=state Circuits Conference (ISSCIRC 2002)*, 403–406, 2002.

[32] Tiri, K. “A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation”. *Proc. of Design Automation and Test in Europe Conference (DATE 2004)*, 246–251, February 2004.

[33] Trappe, Wade, Washington, Lawrence C. *Introduction to Cryptography with Coding Theory.* Prentice Hall publishing, 2006.

[34] Trichina, E. “Combinational logic design for AES subbyte transformation on masked data”. *Cryptology eprint archive: Report 2003/236, IACR*, Nov 2003.

[35] Yang, S. “Power Attack Resistant Cryptosystem Design: A Dynamic Voltage and Frequency Switching Approach”. *Proc. of Design Automation and Test in Europe Conference (DATA 2005)*, 351–365, 2005.

[36] Zabala, Enrique. “Rijndael Cipher”. *Universidad ORT, Montevideo, Uruguay.*

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*
24-03-2011

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From – To)*
Aug 2009- Mar 2011

4. TITLE AND SUBTITLE

Software and Critical Technology Protection Against Side- Channel Analysis Through Dynamic Hardware Obfuscation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

John R. Bochert, 1Lt USAF; john.bochert.1@us.af.mil

5d. PROJECT NUMBER
ENG 10-326

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/11-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Robert L. Herkoltz
Program Manager - Information Operations and Security
Air Force Office of Scientific Research (AFOSR/RSL)
875 N. Randolph Street, Suite 325, Room 3112
Arlington, VA 22203-1768
(703) 696-6565; robert.herkoltz@afosr.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR/RSL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT
Side Channel Analysis (SCA) is a method by which an adversary can gather information about a processor by examining the activity being done on a microchip though the environment surrounding the chip. Side Channel Analysis attacks use SCA to attack a microcontroller when it is processing cryptographic code, and can allow an attacker to gain secret information, like a crypto-algorithm's key. The purpose of this thesis is to test proposed dynamic hardware methods to increase the hardware security of a microprocessor such that the software code being run on the microprocessor can be made more secure without having to change the code. This thesis uses the Java Optimized Processor (JOP) to identify and fix SCA vulnerabilities to give a processor running RSA or AES code more protection against SCA attacks.

15. SUBJECT TERMS
Side Channel Analysis, Cryptographic Processors, Microcontroller Security, Encryption

16. SECURITY CLASSIFICATION OF:

REPORT	ABSTRACT	c. THIS PAGE
U	U	U

17. LIMITATION OF ABSTRACT
UU

18. NUMBER OF PAGES
158

19a. NAME OF RESPONSIBLE PERSON
Dr. Yong C. Kim (ENG)

19b. TELEPHONE NUMBER *(Include area code)*
(937) 785-3636, x4620; Yong.Kim@afit.edu